1998

Synthesis of multiplexor-based FPGAs using 123 Decision Diagrams.

Anthony E. Armah
University of Windsor

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Synthesis of Multiplexor-Based FPGAs using 123 Decision Diagrams

by
Anthony Armah

A Thesis
Submitted to the Faculty of Graduate Studies and Research through the School of Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Science at the University of Windsor

Windsor, Ontario, Canada
1998

ABSTRACT

Field -Programmable Gate Arrays (FPGAs) are programmable devices that can be directly configured by the end user. Reduced Ordered Binary Decision Diagrams (ROBDD) have been used for the synthesis of multiplexor-based FPGAs. However, ROBDDs are extremely sensitive to variable ordering and require all branches to have the same ordering.

In this thesis a new decision diagram based model, the 123 Decision Diagram (123DD), that relaxes some of the constraints of ROBDDs is used to synthesize multiplexor-based FPGAs. 123DD combines both the advantages of ROBDD and BDD.

This thesis focused on the widely used ACTEL FPGAs with emphasis on the general ACT cell.

To my parents, brothers and sisters &
Arlene Conyers

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to Dr. Arunita Jaekel whose tremendous support, tutoring and guidance led to the successful finish of this thesis. What a tremendous and inspiring supervisor she is. My thanks also goes to my committee members Dr. Subir Bandyopadhyay and Dr. H. Kwan for their comments and suggestions regarding this thesis. I would like to thank Mr. Wallid Mnaymneh our system administrator for the help throughout the course of this work. Also my thanks to our secretaries Mary Mardegan and Margaret Garabon who have always been of help to me during my studies here. Also to the rest of the faculty I wish to thank you all for the various support and help given me. I would also like to thank my fellow graduate students for their help and encouragement throughout all these years.

I would also like to thank my mom and dad and my brothers and sisters for their constant support and words of encouragement. Also I would like to thank all my friends for their support. Finally, I must thank Arlene Conyers for the patience, help and support that she gave me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

1.1 Introduction

Over the last few decades, rapid technology changes have caused digital circuit design to undergo several evolutions. Circuit components have evolved from individual transistors to very-large-scale integrated circuits(VLSI). Logic synthesis is no longer limited to assembling discrete components to perform simple logic functions. The design process is becoming increasingly complex as the functionality of the circuits increase dramatically. The use of Hardware Description Languages (HDLs), such as VHDL and Verilog, is becoming an attractive solution for handling the increased complexity in the design process. Automated logic synthesis tools are also available to create circuits that can be mapped onto various devices.

The advances in VLSI technology have opened the door to the implementation of powerful digital circuits at low cost and also

brought very complex digital systems onto a very small area of silicon. Unlike previous generations of hardware technology, in which board level designs included large number of small-scale-integration (SSI) chips that contained basic gates. virtually every digital design produced today consists of a device that contains a high amount of logic per unit area. This is true not only of custom devices such as processors and memory but also of logic circuits such as machine controllers, counters, registers, and decoders. When such circuits are destined for high-volume systems, designers often map them on to high-density gate arrays.

With the rapid changes in technologies and VLSI design techniques, the life cycle of modern products is becoming shorter and shorter. However, the non-recurring engineering (NRE) costs associated with this existing VLSI technologies are still very high. NRE is the cost of the one time initial engineering work. That is the cost first buyer(s) pay, but not the second. Therefore, a design style that will support rapid protyping at almost no or very nominal NRE costs is needed. Field-Programmable Gate Arrays (FPGAs) have emerged as an ideal solution. They have the potential of matching both the speed and density of mask programmed gate arrays (MPGAs). But their manufacturing time and prototype costs are much less than those for MPGAs. FPGAs can be directly configured by the end user, without the use of an integrated circuit fabrication facility.

1.2 Motivation

When faced with the task of designing a logic circuit, designers first come up with a system description of the circuit. The designer then produces an implementation that

meets the design objectives. Finally, the product has to go through a design-fabrication-test cycle. This last phase can sometimes take a year or two or even more. In order to minimze delays in the manufacturing time it is necessary to shorten the design-manufacture-test cycle as much as possible. One way of doing so is to use programmable hardware. The components of this hardware (logic blocks and interconnect) lie uncommitted on an already fabricated chip and can be programmed by the user to implement any kind of digital circuit. This method thus eliminates the manufacturing process from the design process. This speeds up the design process and since the hardware is reprogrammable, it allows the design process to be changed with very little added expense.

Now that the manufacturing step can be bypassed, the actual design of the circuit, which has traditionally been a manual process, can become a bottleneck. With the growing complexity of digital circuits, doing a design completely manually is cumbersome and slow. Therefore automated synthesis tools that start with the specification of the design and produce a satisfactory implementation on the programmable device are required.

1.3 Problem Outline

The problem of synthesis for a popular class of Programmable Gate Array architectures - the multiplexor-based (MB) architectures- has been the focus of many researchers. There are currently several techniques that try to minimize the number of basic blocks that are used to implement a given circuit. These include *Dagon* [20] , which uses a library-based

approach, *Amap*[21] which uses *if-then-else*(ITE) dags, *Prosperine* [22] which uses boolean matching and *mis-pga*[23] which uses **Binary Decision Diagram** (BDD).

In this thesis we will describe a new decision diagram based model for representing Boolean functions. We will then discuss how this model can be used to automatically synthesize arbitrary logic functions and map them onto multiplexor-based combinational logic blocks for FPGAs. In particular, we will be dealing with the well known family of Actel cells for the technology mapping portion of our synthesis algorithm.

1.4 Thesis Organization

Chapter 2 discusses the relevant background material needed for this thesis. This chapter introduces Field-Programmable Gate Arrays (FPGAs) and discusses various applications and programming technologies for FPGAs. It also describes the details of the multiplexor-based generalized ACT cell which will be used in the remainder of the thesis. Finally, Chapter 2 gives an overview of the existing synthesis techniques for FPGAs. This includes various logic synthesis procedures using **Binary Decision Diagrams** (BDDs) and **Reduced Ordered Binary Decision Diagrams** (ROBDDs) and several approaches of technology mapping.

Chapter 3 introduces **123 Decision Diagram** (123DD). It describes the rules for constructing and manipulating a 123DD to synthesize a logic function. This chapter also introduces the rules for transforming an ordered 123DD into an unordered BDD which can be mapped to the general Act cell.

In chapter 4 we discuss the experimental results for our 123DD based synthesis technique. We test our synthesis procedure on a large number of benchmark circuits and compare our results with those obtained from existing ROBDD based design techniques.

Finally in chapter 5 we give a brief conclusion to this thesis work and also give the directions for future work.

Chapter2

Review of FPGAs

2.1 Introduction

This chapter reviews the background materials as well as the major topics needed in this thesis. Section 2.2 focusses on the evolution of FPGAs. Section 2.3 gives an overview of FPGAs, including their architectures, applications, advantages and limitations and programming technologies. Finally section 2.4 outlines the current techniques for synthesis of multiplexor based FPGAs. In particular it looks in detail at decision diagram based synthesis techniques and concludes with an overview of several approaches to technology mapping.

2.2 Evolution of FPGAs

Field Programmable Devices are general-purpose chips that can be configured in a wide variety of applications [3]. The first of the series of programmable devices was Programmable Read-Only Memory (PROM). A PROM is a one-time programmable device

that consists of read-only cells. It implements a logic circuit by using its address lines as the circuit's inputs. The outputs of the circuits are defined by the stored bits. Two versions of PROMs that exist are *Mask-programmable* which can be programmed only by the manufacturer and *Field-programmable* that can be programmed by the end user. Mask-programmable chips have a better speed performance during logic circuit implementation than field-programmable because connections in mask-programmable devices are hardwired during manufacture. In field-programmable devices, connections involve programmable switches which are slower in speed. On the other hand, field-programmable chips are less expensive and can be programmed immediately.

Enhancement of the programmability feature of PROMs led to devices such as *Erasable Programmable Read-Only Memory* (EPROM) and *Electrically Erasable Programmable Read-Only Memory* (EEPROM) which can be erased and reprogrammed to be developed.

Another type of programmable device designed for logic circuits is the *Programmable logic Device* (PLD). PLDs consist of an array of **AND** gates connected to an array of **OR** gates. The basic type of PLD is *Programmable Array Logic* (PAL). PAL consists of a programmable AND-plane followed by a fixed (non-programmable) OR-plane. Due to the non-programmable OR-plane, PALs are considered to be less flexible. Then a more flexible type of PAL which has a programmable AND-plane and a programmable OR-plane was developed and called a *Programmable Logic Array* (PLA). PLAs are once again available in both mask-programmable and field-programmable versions.

The introduction of PAL and PLA devices lead to the developments of devices such as *Simple programmable-logic devices* (SPLDs), and *Complex programmable-logic devices* (CPLDs). SPLDs are a collection of PLAs, PALs and PAL-like devices into a single programmable-logic device. They are low cost and have very high speed performance. The drawback with SPLDs is that they have limited logic capacity and cause problems as the number of inputs increases. The only way to provide a large-capacity device based on SPLD architecture is to interconnect multiple SPLDs on a single chips. This is called CPLD [2].

CPLDs provide logic capacity up to the equivalent of about 50 SPLDs. But difficulties occur when the architecture is extended to higher densities [2]. Thus building FPDs with very high logic capacity required a different approach. High capacity general-purpose logic chips were then developed and they are called *Mask-programmable Gate Arrays* (MPGAs). An MPGA consists of rows of transistors that can be interconnected to implement a desired logic circuit. User specified conditions are available both within the rows (to implement basic logic gates) and between the rows (to connect the basic gates together). In MPGA, all the mask layers that define the circuitry of the chip are pre-defined by the manufacturer, except those that specify the final metal layers. MPGAs are very expensive and the turnaround time is very long as they have to go through the metallization steps in a fabrication facility. The development of MPGAs motivated the design of field-programmable equivalents which are called *Field-Programmable Gate Arrays* (FPGA). The stages of the evolution of PLDs is shown in Figure 2.1. The ellipses represent the PLDs. PROMs are not PLDs, because they are mostly used only for memory

applications. MPGAs are also not PLDs, because they can be configured only during chip fabrication.

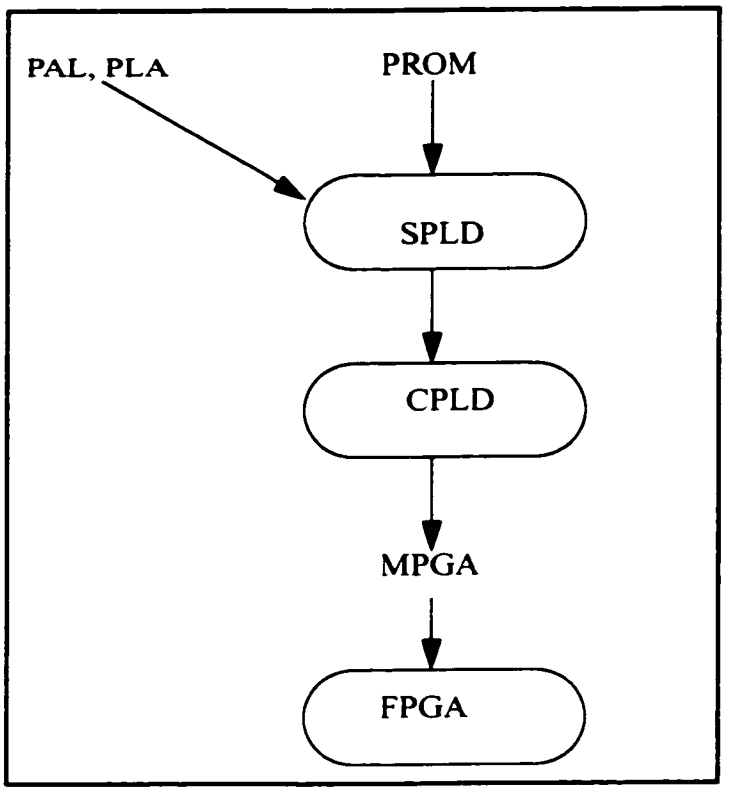

Figure 2.1 The evolution of FPGAs

2.3 Overview of FPGAs

FPGAs are the result of a successful marriage between MPGAs and PLDs. They allow implementation of digital designs in multilevel logic structures and are fully user programmable.

An FPGA consists of an array of uncommitted logic and interconnections. The interconnections can be programmed to configure an FPGA to implement any desired digital function. The user-programmability and the ease of designing using FPGAs is achieved at the cost of circuit layout density. FPGAs have to accommodate programmable switches and wires for routing. This makes the chip level architecture of FPGAs very restrictive. Figure 2.2 shows a typical FPGA architecture. This consists of a two-dimensional array of logic blocks that can be connected by a general interconnection resource. These interconnections are made of segment wires which may be of various lengths. The logic circuits are implemented by partitioning the logic into individual logic blocks and interconnecting the blocks as required via the switches. The structure and content of a logic block are called its *architecture.* There are different types of logic block architectures available and these are built using look-up tables, multiplexers or PALs.

The structure and content of the interconnect in an FPGA is called its *routing architecture* which consists of wire segments and programmable switches. The programmable switches can be constructed in several ways, including *pass-transistors* controlled by *static RAM cells*, *anti-fuses*, *EPROM transistors*, and *EEPROM transistors* [3] There exist different ways to design the structure of the routing architecture. Some FPGAs offer simple connections between blocks, and others provide fewer, but more complex routes.

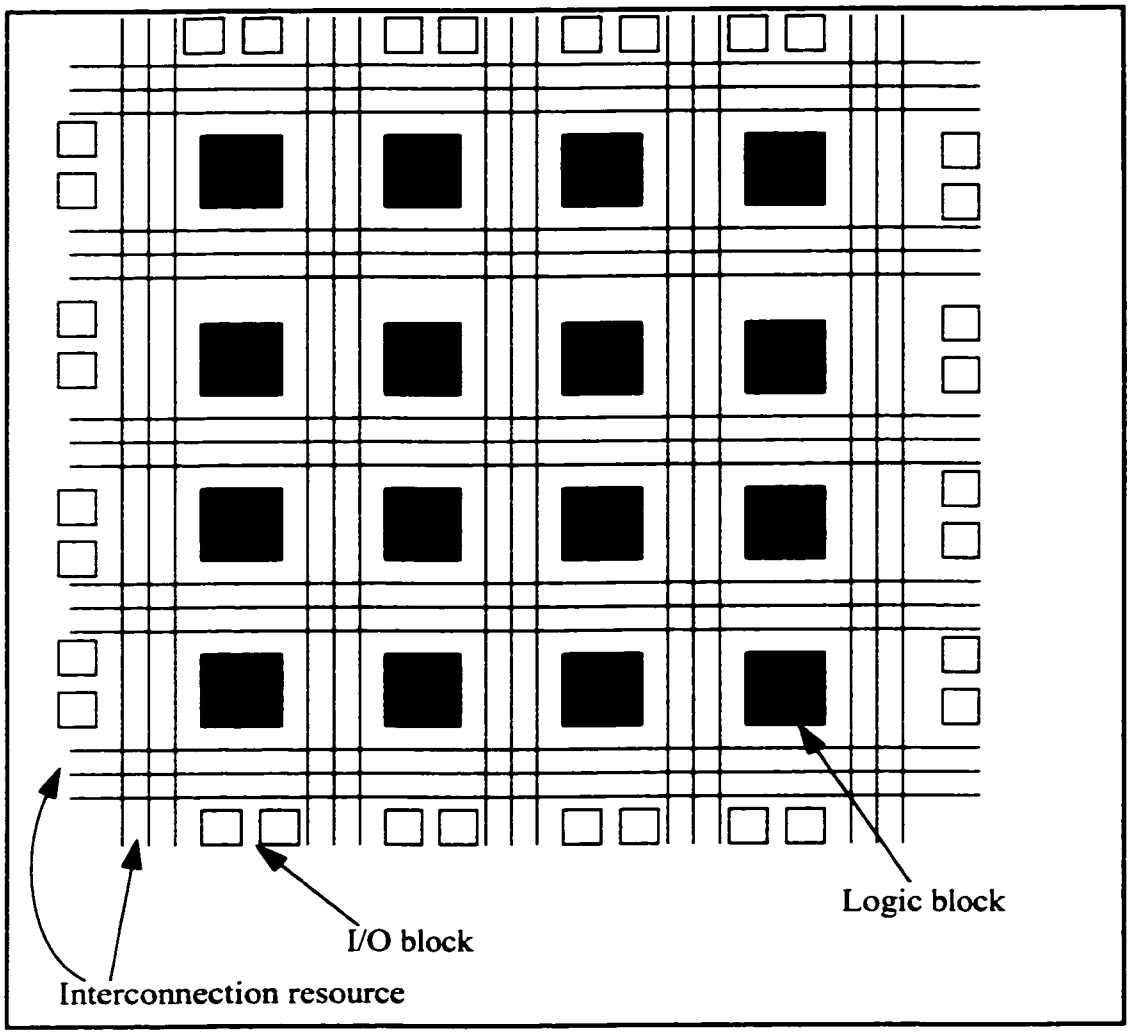

Figure 2.2 : FPGA Architecture

2.3.1 Applications of FPGAs

FPGAs can be used in almost all of the applications that currently use MPGAs, PLDs and small scale integration (SSI) logic chips. Some of the applications are listed below [3]

Application-Specific Integrated Circuits(ASICs): An FPGA is a general medium for implementing digital logic and is particularly suitable for implementation of ASICs. Some examples are 1megabit FIFO controller, an IBM PS/2 micro channel interface, a DRAM

controller with error connection, a graphics engine, and an optical character recognition circuit.

Implementation of Random Logic: PALs are usually used to implement Random Logic circuitry. If the speed of the circuit is not of critical concern (PALs are faster than most FPGAs), such circuitry can be implemented advantageously with FPGAs. A single FPGA can implement a circuit that can require ten to twenty PALs.

Replacement of SSI Chips for Random Logic: A single FPGA can replace a number of SSI chips in existing commercial products, which results in substantial area reduction of the circuit boards that carry such chips.

Prototyping: FPGAs are well suited for prototyping logic designs. The low cost of implementation and the short time needed to physically realize a given design, provide them with enormous advantages over traditional approaches for building prototype hardware.

FPGA-Based Compute Engines: FPGAs inspired a whole new class of computers. These computers consists of a board in-circuit re-programmable FPGAs with interconnections mostly between neighboring chips. The idea is that a software program can be "compiled" (using high-level, logic-level synthesis) into hardware rather than software. The hardware is then implemented by programming the board of FPGAs. This approach has two major advantages: first, there is no instruction fetching as requires by traditional microprocessors, as the hardware directly embodies the instructions. Secondly, this

computing medium provides high levels of parallelism, resulting in further speed increase. With this approach and at the research level, performance achievement ranging from 25 billion operations per second up to 264 billion operations per second on applications such as RSA cryptography, the discrete cosine transform and 2-D convolution has been achieved by Digital Equipment Corporation in Paris.

On-Site Re-configuration of Hardware: Some FPGAs can be reprogrammed unlimited numbers of times. Reprogrammability is a very attractive feature where hardware has to be changed dynamically, or where hardware has to be adapted to different user applications. An example is a computer equipment in a remote location whose hardware has to be altered on site in order to correct a failure or perhaps a design error.

2.3.2 Advantages and Limitations of FPGAs

Faster Design and Verification: FPGAs can be designed and verified in a few days, while the same process requires several weeks with gate arrays. There are no non-recurring engineering (NRE) costs and no prototypes to wait for.

Design Changes without Penalty: Because the devices are software configured via instant programming, modifications are much less risky and can be made anytime, in a matter of hours instead of the weeks it would take with a gate array. This adds up to significant cost savings in design and production.

Shortest Time to Market: Designing with programmable logic, time-to-market is measured in days or a few weeks, rather than the months required when designing with gate arrays.

The two major disadvantages of FPGAs are their relatively low speed of operation, and relatively low logic density (compared to MPGAs). The propagation delays in FPGAs is adversely affected by the inclusion of programmable switches, which have significant resistance and capacitance, in the connections between logic blocks. A direct comparison with MPGAs indicates that a typical circuit will be slower by a factor of roughly three if implemented in an FPGA. Logic density is decreased because the programmable switches and associated programming circuitry require a great deal of chip area compared to the metal connections in an MPGA. Typical FPGAs are a factor of 8 to 12 times less dense than MPGAs manufactured in the same IC fabrication process. Thus at high production volumes FPGAs becomes much more expensive than MPGA [3].

2.3.3 Programming Technologies

Programming technology paves the way for a better understanding of how FPGAs are made to be field-programmable. The term *programmable switch* refers to the programmable elements of the FPGA chip. The elements are used to implement the programmable connections among the FPGAs logic blocks. A typical FPGA may contain more than 100,000 programmable elements. These programming elements are implemented using different technologies, such as static *RAM (SRAM) cell, anti-fuse, EPROM transistors and EEPROM transistors.* Whatever the technology of implementation, the programming elements all share the property of being configurable in one of two states: ON or OFF. The desirable properties of the programmable elements are:

- The chip area consumed by the programmable elements should be as little as possible.

- They should have a low ON resistance and a high OFF resistance.
- The programmable element should contribute low parasitic capacitance to the wiring resources to which it is attached,
- It should be possible to reliably fabricate a large number of programmable elements on a single chip [3].

Although no technical reason prevents application of EPROM or EEPROM to FPGAs, current commercial products use SRAM or anti-fuse technologies.

2.4 Existing Approaches to FPGA Synthesis

The synthesis process for FPGAs is separated into two phases:

- a technology-independent optimization phase which is also called *logic optimization*
- a *technology mapping* phase where the optimized logic circuit is mapped to the target architecture.

Logic synthesis is the process whereby optimized logic-level representations are generated from a high-level description. The optimized logic-level is the interconnection of logic gates and the high-level description is the original network or Boolean function. Logic synthesis is of great importance because, the growing advances in integrated circuit technology lead to a corresponding increase in design complexity. Automated design tools are thus becoming essential in order to shorten the time of design while at the same time achieving the best performance.

Figure 2.3 shows the stages involved in logic synthesis. As shown in the figure, the optimization phase is where an attempt is made to generate the representation of the circuit. In the second phase (Technology mapping) the optimized representation is transferred into the final circuit.

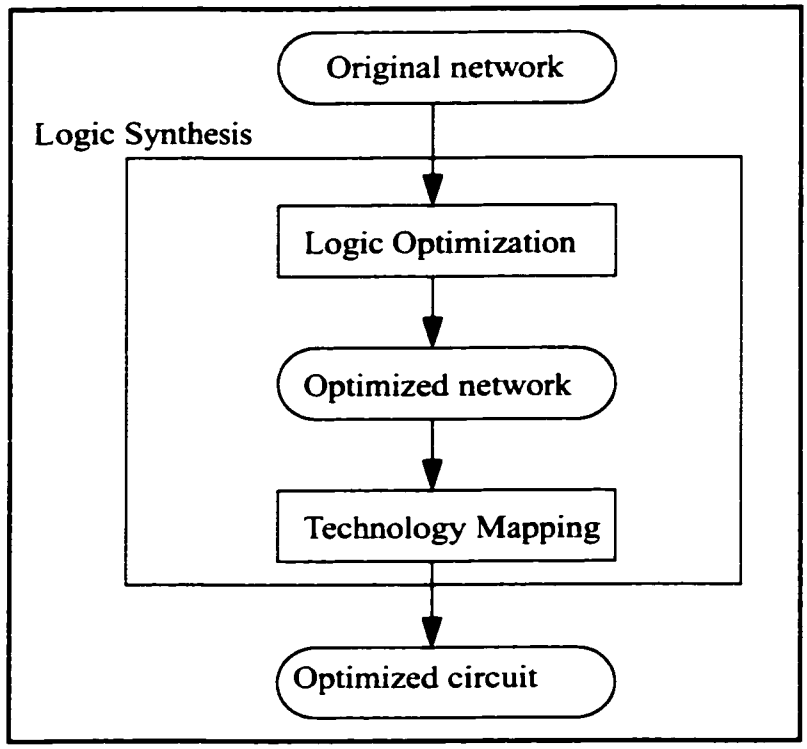

Figure 2.3 The stages of Logic Synthesis

The technique used for the logic optimization is *decision diagram* based. This is particularly suitable for synthesis of multiplexor-based FPGAs. Since our research focuses on this class of FPGAs, we will look at decision diagram based synthesis techniques in more detail in the following section.

2.4.1 Decision Diagram Based Synthesis

A *binary decision diagram* (BDD) is a type of graph used notably for the description of algorithms. It assemblies two types of nodes: *the decision or test node* and the *output node*. The decision node is equivalent to an *if-then-else* instruction: it realizes a test on a binary variable and, according to this value, indicates the node following. The output node produces a value. There are two rules of assembling of BDDs, these are:

- there is one and only one initial node (the entry point of the algorithm);
- the output point of a node can be connected to only one entry point of another node.

It has been demonstrated that all logical boolean functions can be represented by a binary decision diagram [5]. The function of a decision node can be implemented by a multiplexor circuit, and a binary decision diagram can be implemented by an interconnection of these circuits. In all this cases, the minimalisation of the number of nodes used is important, for the cost and/or the time of execution of the function. Nevertheless, the complexity of this minimalisation is such that in most cases approximate solutions are accepted [6].

All decision diagrams are extremely sensitive to variable ordering. BDDs are attractive because they allow flexibility in the ordering of input variables. However, they have the drawback of not allowing sharing among their branches and this causes replication of functions in different branches. Figure 2.4 shows an example of a function, $f = acd + \bar{a}bd + ac\bar{d} + b\bar{c}\bar{d}$, represented by a BDD.

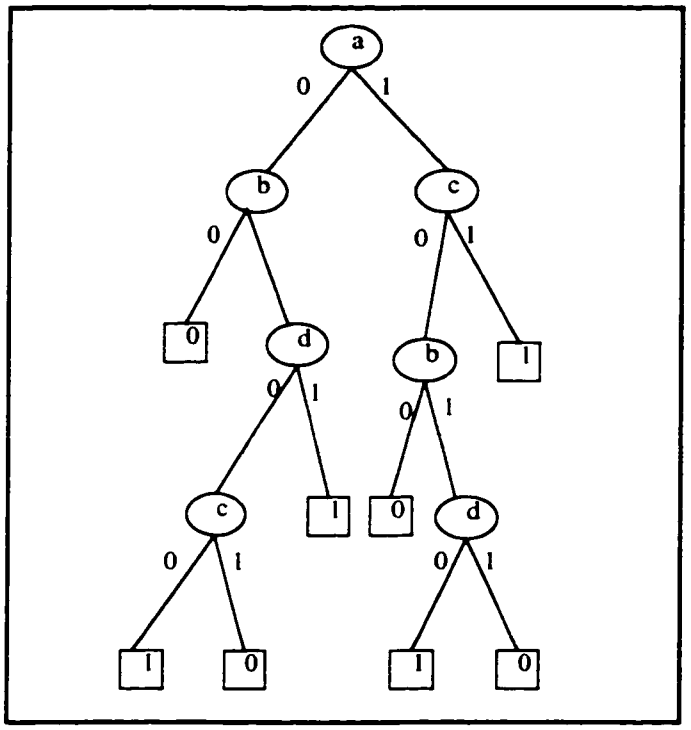

Figure 2.4 A BDD represented by the function f

An ***Ordered BDD*** (OBDD) is a BDD whose labels along every directed path occur in the same order and no label appears more than once in each path. A ***Reduced OBDD*** (ROBDD) is an OBDD where each node represents a distinct logic function. The ROBDD representation of a Boolean function is unique for a given variable ordering and therefore constitutes a canonical representation of the function. In ordinary terms, ROBDD can be said to be a special form of BDD. ROBDDs allow sharing of subfunctions but are sensitive to variable ordering. The input ordering is a very strong optimization criterion for reducing the complexity of the ROBDD representations. Figure 2.5 shows the ROBDD for the function $g = ab + de$ with two different variable orderings a,d,b,e and a,b,d,e.

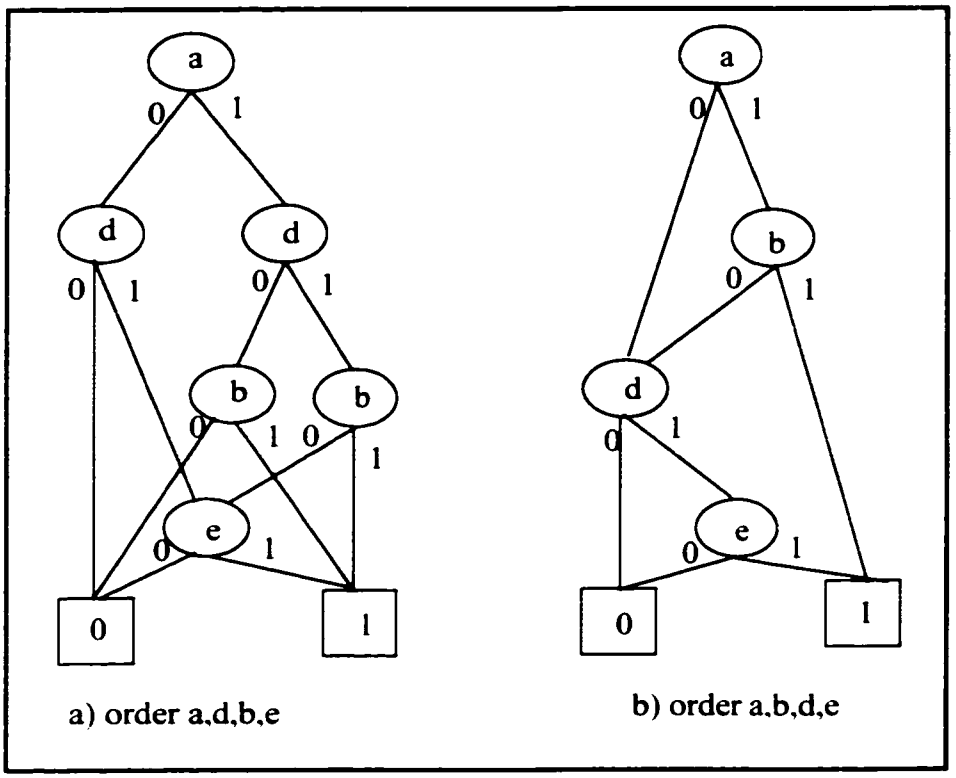

Figure 2.5 ROBDD of function g for two variable ordering

2.4.2 Technology Mapping

Technology mapping is a step that maps the logic designs on FPGA logic blocks. The output of technology mapping is an implementation of digital design that is functionally equivalent to the original design but physically composed of FPGA logic blocks. Technology mapping is done by selecting pieces of the whole of the network and implementing it on one of the available circuit elements. Conventionally, technology mapping has focused on using circuit elements from a limited set of simple gates, such as a standard cell library. The next sections describes technology mapping programs that are commonly used for FPGAs.

2.4.2.1 Lookup Table (LUT) Technology Mapping

The basic block of an LUT architecture is a look-up table that can implement any boolean function of up to m inputs, $m \geq 2$. For a given LUT architecture, m is a fixed number. In commercial architectures, m is typically between 3 and 6. The basic unit of logic in these architecture is called a **configurable logic block (CLB)**. A typical architecture is the Xilinx architecture in which m is 5. The interconnections between the logic blocks consist of metal segments and the interconnection is joined by program-controlled pass transistors. The logic functions and the interconnections are determined by the configuration program data stored in the internal static memory cells. The constraints with type of architecture are a limited number of CLBs on a chip, the maximum number of inputs a CLB can have and the limited wiring resources. A typical program that uses LUT is MisII [24] that uses the number of variables in the factored form of a node in the Boolean network as an estimate of the area or the gate count. For example, let m = 5, $f_1 = abcdeg$ and $f_2 = abc + \bar{b}de + \bar{a}\bar{e} + \bar{c}\bar{d}$. Functions f_1 and f_2 have 6 and 10 literals respectively. However, f_1 requires two CLBs in its implementation, whereas f_2 needs just one (since f_2 is a function of five variables). Thus the objective function has to do more with the number of inputs than with the actual logic that the function realizes.

2.4.2.2 Multiplexor-Based (MB) Technology Mapping

In MB architectures, the basic block is a configuration of multiplexers. MB architectures can be of many different kinds. In this thesis, the Actel architecture, shown in Figure 2.6, is chosen as the primary target. The basic combinational logic block is the Act1 cell (Figure 2.6b) which has a configuration of three 2-to-1 multiplexors, with an OR gate

providing the select input to the output multiplexor. For simplicity sake, this research is based on a simplified, more uniform structure: the generalized Act cell (Figure 2.6a) where one of the inputs of the OR gate is tied to zero.

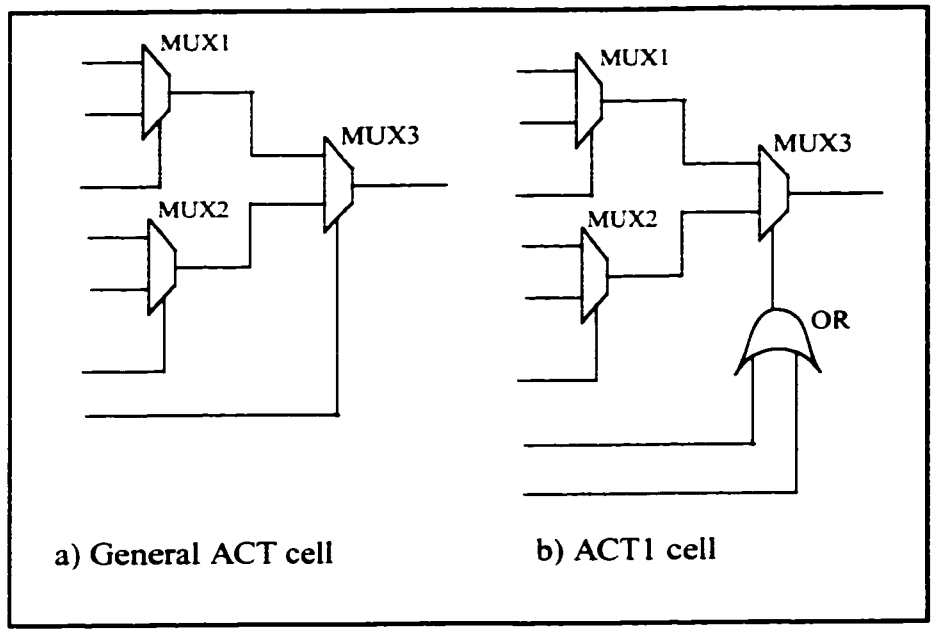

Figure 2.6 Actel Architectures

One MB logic block can implement a large number of different functions. It can implement all 2-input functions, all 3-inputs functions and several functions with more inputs. Some examples of technology mappers for MB logic blocks are mis-pga [23], Proserpine [22], Amap [21], and Xmap [25]

2.5 Conclusions

In this chapter we have discussed the relevant background material for synthesis of multiplexor-based FPGAs. Existing approaches generally use ROBDDs for logic

optimization followed by a technology mapping phase. In the remainder of the thesis we will introduce a new technique for FPGA synthesis which provides considerable improvement over existing methods.

Chapter 3

123Decision Diagram model

3.1 Introduction

The increasing use of programmable devices is one of the most important developments in the ASIC world for the present decade. Due to their high flexibility and fast turn around they constitute an ideal support for fast prototyping as well as for actual designs. In this chapter we focus on the *123Decision Diagram* (123DD) model which can be used to synthesize a well-known class of programmable devices - *multiplexor-based FPGAs*. In this chapter, we discuss the main characteristics as well as the transformation rules for manipulating 123DDs. We also give an algorithm for automatically synthesizing multiplexor-based FPGAs, using the 123DD and describe the technology mapping phase, which maps the final 123DD onto the generalized ACT cell. Finally, we give an example which illustrates all the necessary steps in the synthesis procedure.

3.2 123Decision Diagram (123DD)

In this section we describe a model for representing circuits called the 123 decision diagram(123DD). This model was introduced in [9] and we will briefly outline its important features in this section.

123DD is a decision diagram based model, like the ROBDD. However, the 123DD can accommodate certain transformations which are not allowed in ROBDDs. This results in a structure which may have one, two or three edges coming out from a node. An ROBDD, on the other hand, must always have exactly two edges from each node. This flexibility in the structure of 123DDs can lead to more efficient FPGA implementations. The main characteristics of the 123DD model are given below:

- A nonterminal node is shown as a labelled circle. It represents a net in the circuit.
- A node at the i^{th} level has label X_i, denoting the i^{th} input variable.
- Each nonterminal node has 1, 2 or 3 outgoing edges. Each edge has a label of 0, 1 or 2.
- Each edge with a label 0(1) from a node with label X_i corresponds to a transistor with its gate connected to $\overline{X}_i$ (X_i).
- An edge with a label 2 indicates a metal line.
- The root node corresponds to the output.

An example of a 123DD for the function $f = \bar{a}\bar{b}\bar{c} + \bar{a}bd + \bar{a}c\bar{d} + a\bar{b}\bar{c} + acd$ with a variable ordering of a,b,c,d and a bitmap = 1110 0111 1101 0001 is shown in Figure 3.7.

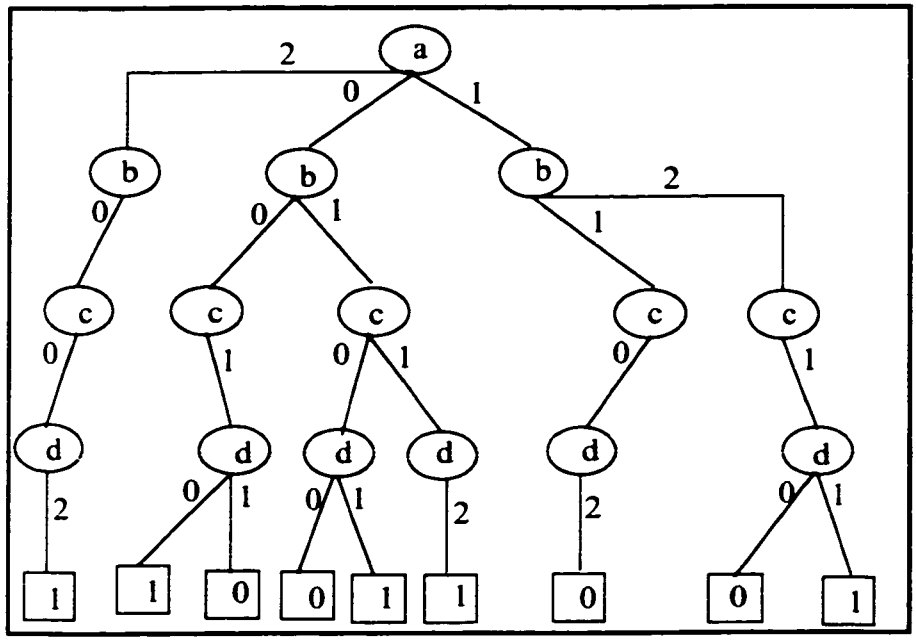

Figure 3.7 A 123 Decision Diagram

3.3 Logic Synthesis using 123DD

The synthesis of 123DD exploits the advantages of both ROBDDs and BDDs i.e it allows sharing of sub-functions (like ROBDDs) and at the same time allows some flexibility in variable ordering (like BDDs). The synthesis process involves the construction of an initial *ordered* 123DD. This means that all branches of the decision diagram are required to have the same variable ordering, There are three rules that are used to construct the 123DD. These are

i) Node Generation rule,

ii) Substitution rule and

ii) Merging rule.

The details of each rule are are described in the following sections.

3.3.1 Node Generation rule

The node generation rule states that from a nonterminal node n at level i and with an associated bitmap b of length $2m$ we can generate two children (left and right child) of n, $n_0(n_1)$ at the $(i+1)^{th}$ level where $n_0(n_1)$ is connected to n through edges labelled 0(1) and have bitmaps $b_0(b_1)$ of length m such that $b_0 \bullet b_1 = b$. Here, the operation $b_0 \bullet b_1$ simply implies the concatenation of the two individual bitmaps b_0 and b_1. This rule is used to generate the next level of the 123DD, once all the nodes at the current level have been processed. It is identical to the corresponding rule for constructing ROBDDs. This rule is applied to each node and at a particular level until all the nodes at the current level are terminal nodes. Figure 3.8 shows an example of the node generation rule.

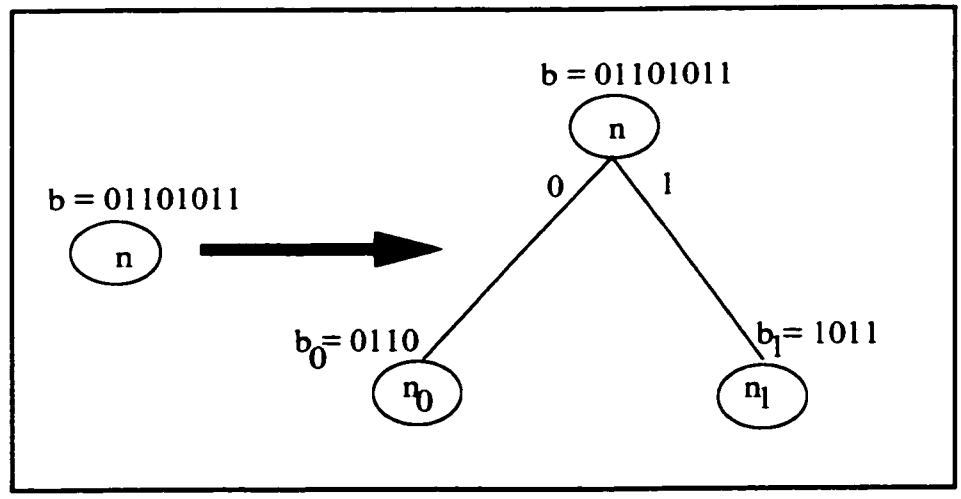

Figure 3.8 Example of the node generation rule

3.3.2 Substitution rule

For this rule, we consider a node n with paths p_1 and p_2 to two nodes n_{k1} and n_{k2} with bitmaps b_1 and b_2 respectively. Except for the common ancestor, if the label of the edges are the same and they have the same bitmap then we replace the two nodes with a single path p_3 with a label 2 from node n to n_{k3}. Again this procedure is repeated for the rest of the tree. Figure 3.9 is an example of the substitution rule.

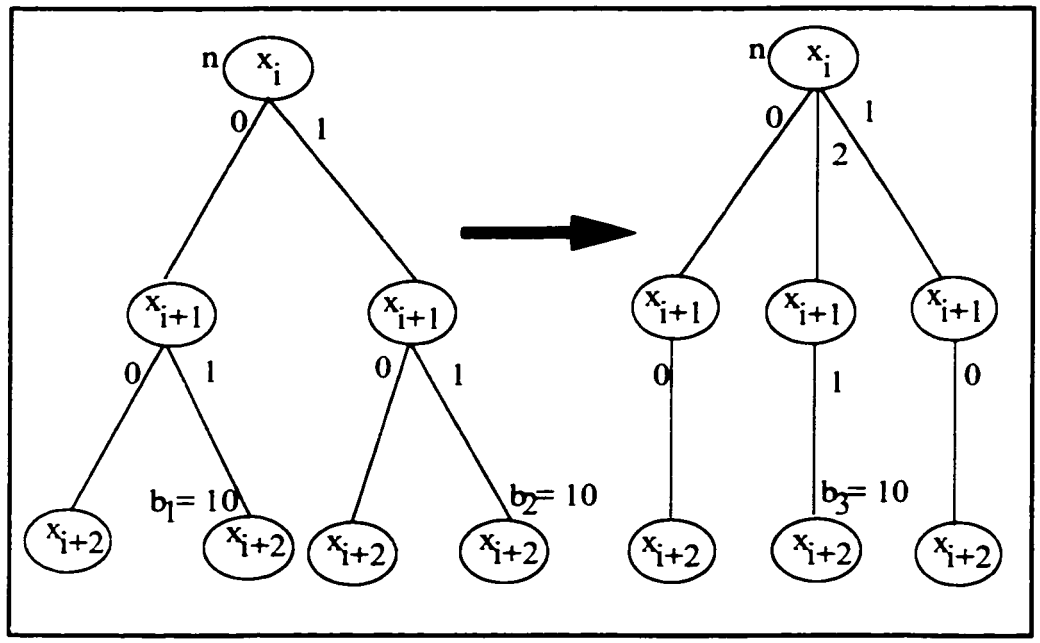

Figure 3.9 Applying the Substitution rule on the 123DD

3.3.3 Merging rule

If the tree contains two or more nodes with the same bitmaps then it implies that those nodes denote the same function. So instead of having to expand each node separately (resulting in duplication), we merge them and one of the nodes is expanded. This thus

reduces the number of nodes in the 123DD. Figure 3.10 explains how to use the merging rule in a 123DD.

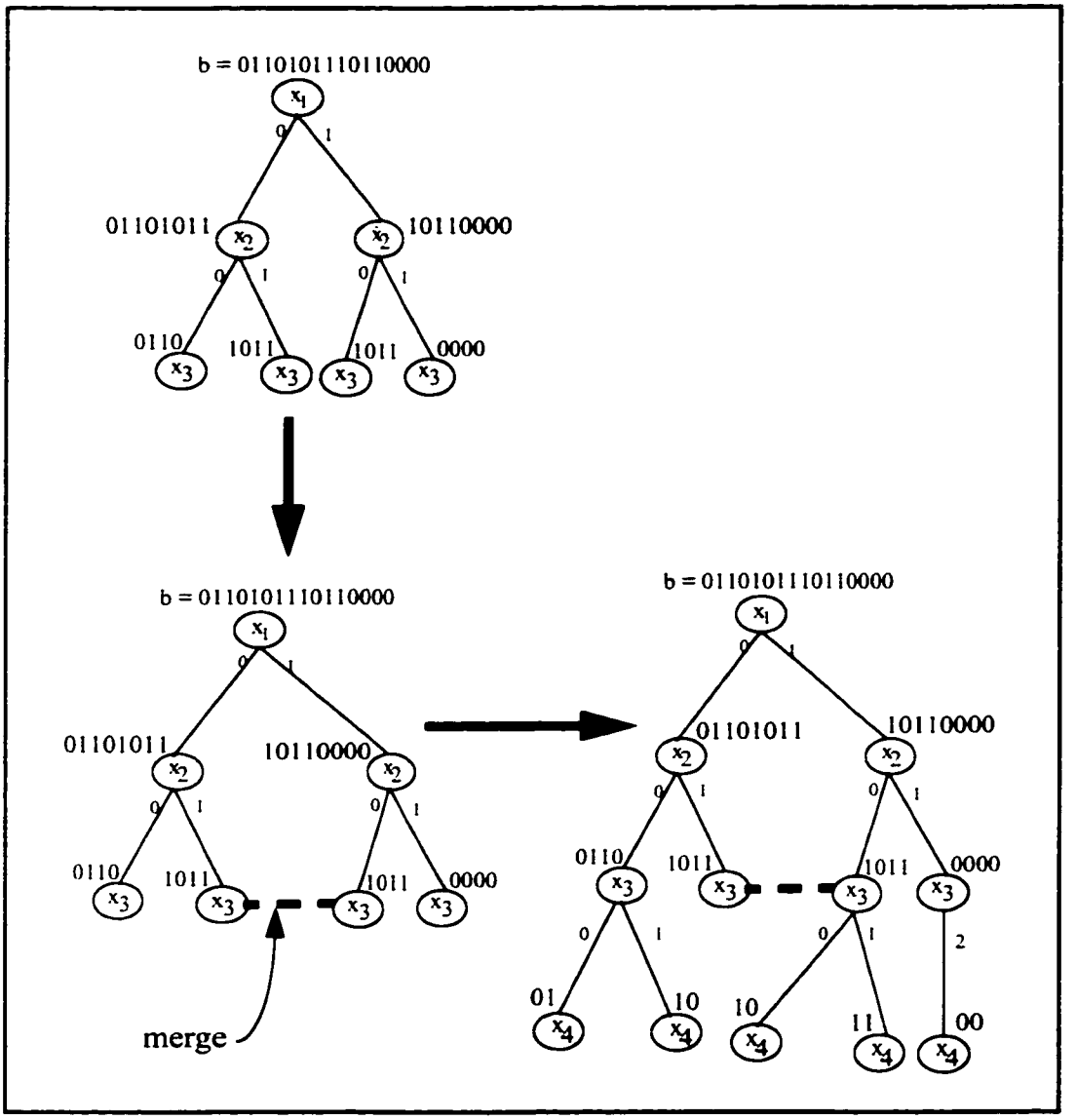

Figure 3.10 Using the merging rule on the 123DD

3.3.4 Outline of Synthesis Procedure

In this section we give an overview of the synthesis procedure used to construct the initial *ordered* 123DD for a given Boolean function f of n variables with a specified variable ordering. Here the function f is defined in the form of a bit map of size 2^n . This is a top-down recursive heuristic which proceeds level by level. Each level corresponds to one input variable (in true or complemented form). The inputs are arranged in order from the most significant (top level) to the least significant (bottom level).

1. Current level = 1

Initialize the current level to 1. Step 2 must be repeated for each level in the 123-DD.

2. REPEAT

- ***a. Generate children.***

  Use node generation rule, to generate two children from each nonterminal node at the current level.

- ***b. Current level = Current level + 1.***

  The next level, consisting of nodes generated in 2a, becomes new current level.

- ***c. Reduce the number of paths in the 123-DD.***

  For each pair of nodes (N_1, N_2) at the current level, combine the two paths from their common ancestor to N_1 and N_2 into a single path by applying the substitution rule, if possible.

- ***e. Merge subgraphs***

  For each pair of nonterminal nodes (N_1, N_2) merge the two nodes into a single node, using the merging rule, if possible

UNTIL *there are no nonterminal nodes at the current level.*

Figure 3.11 Outline of Synthesis Procedure

At each level, the *substitution rule* and *merging rule* may be applied to the nodes to manipulate the graph. When no more transformations can be applied to any of the nodes at the *current level*, the nodes are expanded, using the *node generation* rule, to generate the next level, which becomes the new *current level*. This process continues for each level until all nodes at the *current level* are terminal nodes.

3.4 Transformations on 123DD

The ordered 123DD obtained by the above synthesis procedure is not suitable for directly mapping onto an FPGA architecture. Therefore, we have to apply some transformations on this initial 123DD to obtain an unordered BDD. The goal of such transformations is twofold:

a) Further optimize the 123DD by appropriately changing the variable ordering of certain branches, and

b) Make the final representation suitable for implementation as an interconnection of ACT cells.

These transformations are carried out by looking at each node in the 123DD and identifying the branches where a different variable ordering may be beneficial. We then change, if possible, the variable orderings in these branches. The ordering change takes place only if there is no conflict with the subgraphs shared using the merging rule. A brief overview of this step is shown in Figure 3.12.

1. Set current_node = root_node;

2. Process current_node

a) IF current_node == NULL ***RETURN;***

b) IF current_node has edge with label 2, remove the edge;

c) Process left child of current node;

d) Process right child of current node;

RETURN;

Figure 3.12 : Overview of transformations on 123DD

Each node in the 123DD is processed starting with the root node. If a node does not have an edge with label 2, no transformations are needed and we simply proceed to the next node. If, there is an edge with label 2, we apply some transformations to that node to eliminate the edge before proceeding to the next node. The procedure ends when all the nodes in the 123DD have been processed and there are no remaining edges with label 2.

The final decision diagram model, which is mapped to the multiplexor-based cells should not have any edges with label 2. Edges with label 2 in the 123DD may be removed in one of three possible ways.

(i) If there is only one edge from a node and that edge has label 2, the corresponding node and edge may be deleted from the 123DD (e.g. edge e_2 in Figure 3.13(a)).

(ii) An edge may be removed by choosing a more suitable variable ordering for the

branch containing that edge (e.g. edge e_1 in Figure 3.13(a)).

(iii)An edge may be removed by replacing the branch by two separate branches if a suitable variable ordering cannot be found.

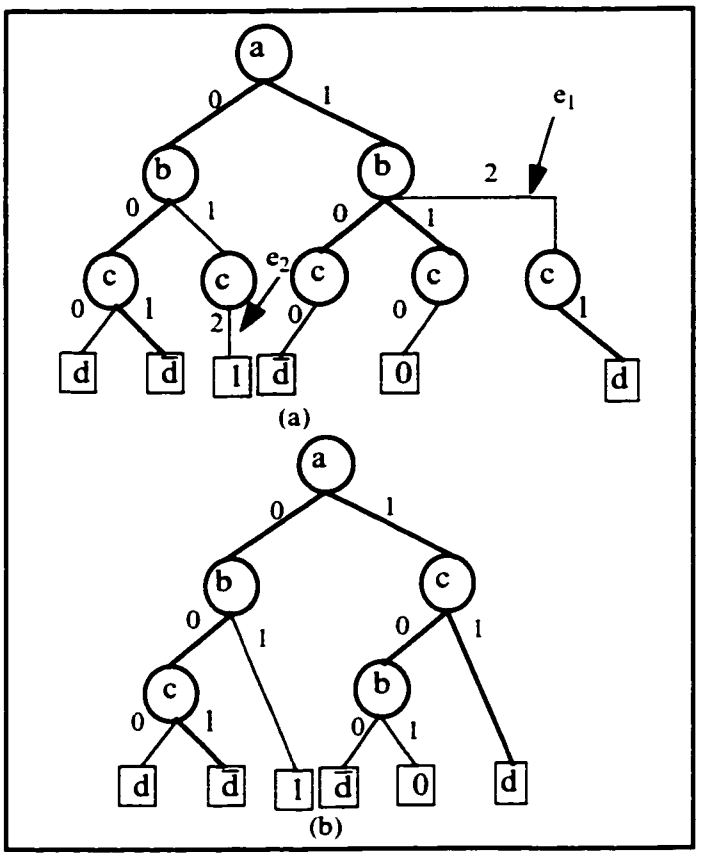

Figure 3.13 : Transformations on 123DD

It is always better to remove edges by the first two methods, since this helps to further optimize the decision diagram. However, if this is not possible, we must use the third method to eliminate an edge with label 2. For example, consider the function with the 123DD shown in Figure 3.13(a). The edge e_1 can be removed by switching variables b and c in the associated branch and edge e_2 may be removed by deleting the corresponding

node with label 2. After these transformations, the final decision diagram (Figure 3.13(b)) is ready for technology mapping.

The transformations discussed in this section systematically remove edges with label 2 from the 123DD. This reduces the 123DD to a BDD with (possibly) different variable orderings on different branches. Thus, we are able to combine the efficiency of ROBDDs, due to sharing of nodes, with the flexibility of BDDs.

The code to implement the above algorithm (named *modify_tree*)is included in the appendix.

3.5 Illustrative Example

In this section we explain the entire logic optimization phase, from the synthesis of the initial 123DD to the final transformations, through a simple example. We consider a four variable function f=1110 0111 1101 0001 specified as a bitmap. Figure 3.14 shows the 123DD at various stages of the synthesis. Figure 3.14 (a), shows the root node with the associated bitmap representing the entire function. Applying the *node generation rule* to this root node, we obtain Figure 3.14(b) showing its children. Application of *node generation rule* to the next two levels generates Figure 3.14(c). At this point, when we check the 123DD to see if any of the other transformations rules can be applied, we see that there are two pairs of paths (I and II) which can be combined using the *substitution rule*. These paths are indicated by the thicker lines in Figure 3.14(c). The *substitution rule* is then applied to I and II and the resulting combined paths (starting with edges e_1 and e_2

respectively) are shown in Figure 3.14(d). Similarly we see that there are two nodes which may be merged at this level using the *merging rule*. Applying these rules to the 123DD we obtain Figure 3.14(d)

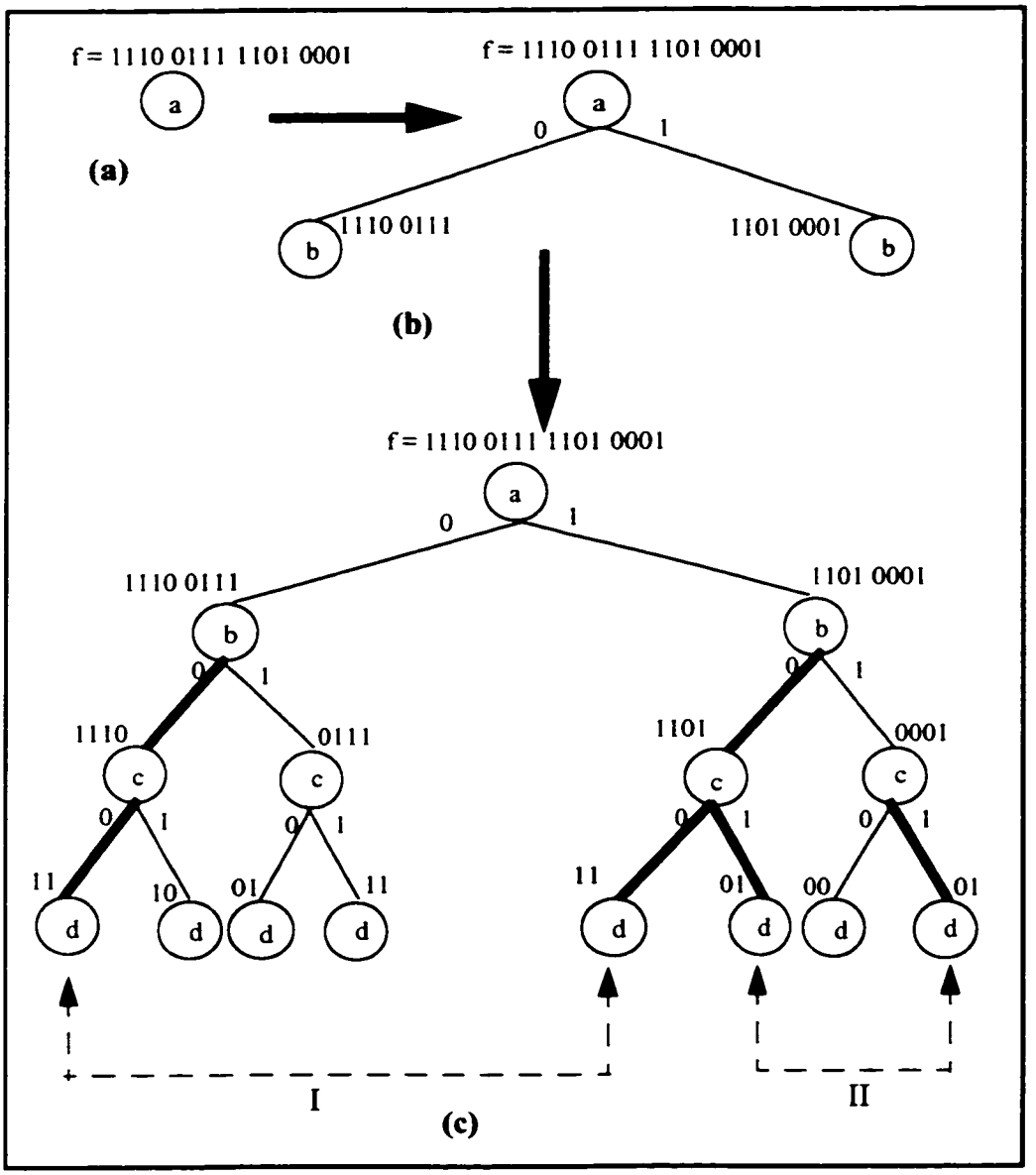

Figure 3.14 : Example of logic optimization

Figure 3.14(d) represents the ordered 123DD, obtained from the initial synthesis procedure. All edges with label 2 (i.e. edges e_1 and e_2) should be removed from this 123DD. Both of these edges may be removed by changing the variable ordering as stated in section 3.4. The edge e_1 is removed first by switching variables **c** and **a.** Then e_2 is removed by switching variables **a** and **b** in the right subtree of the root node. This process, however, creates more edges with label 2 as shown in Figure 3.14 (e).

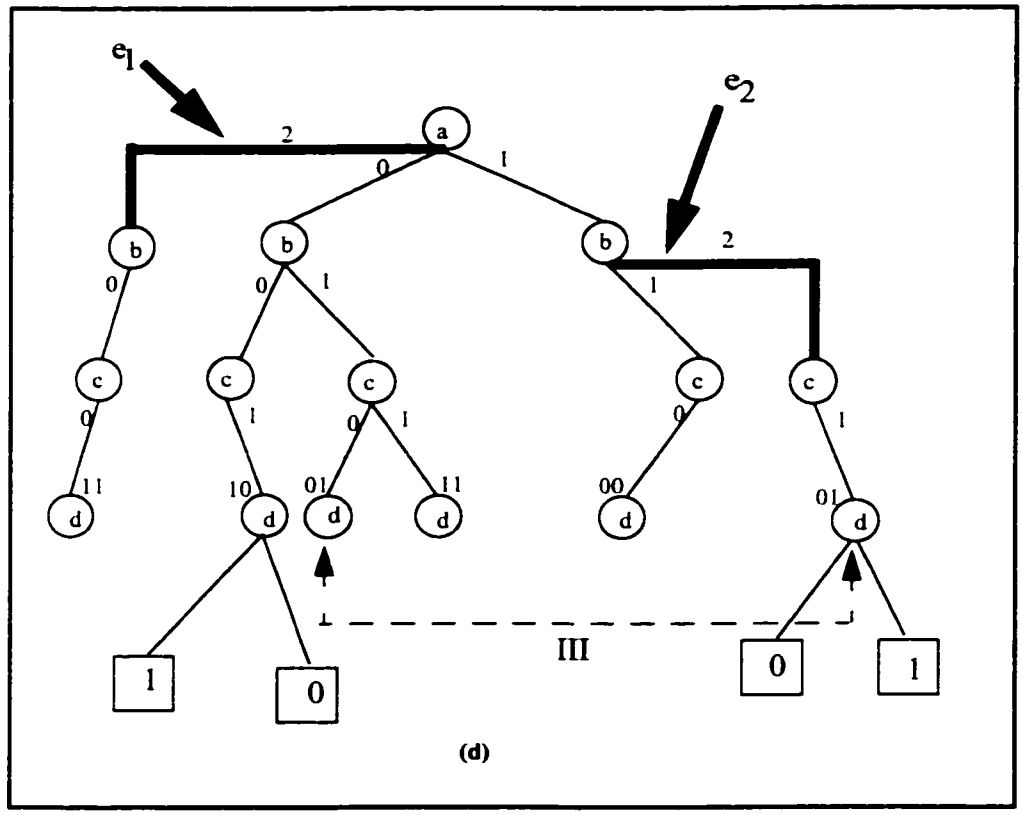

Figure 3.14 : Example of logic optimization

We note that each of these new edges e_3, e_4, e_5, e_6, and e_7 can be removed by using the second rule in section 3.4, i.e. simply deleting the edge and the associated nonterminal

node. The final and optimized 123DD is given in Figure 3.14(f), which can be directly mapped onto the ACT cell. Note that I, II, III, IV and V represents merging.

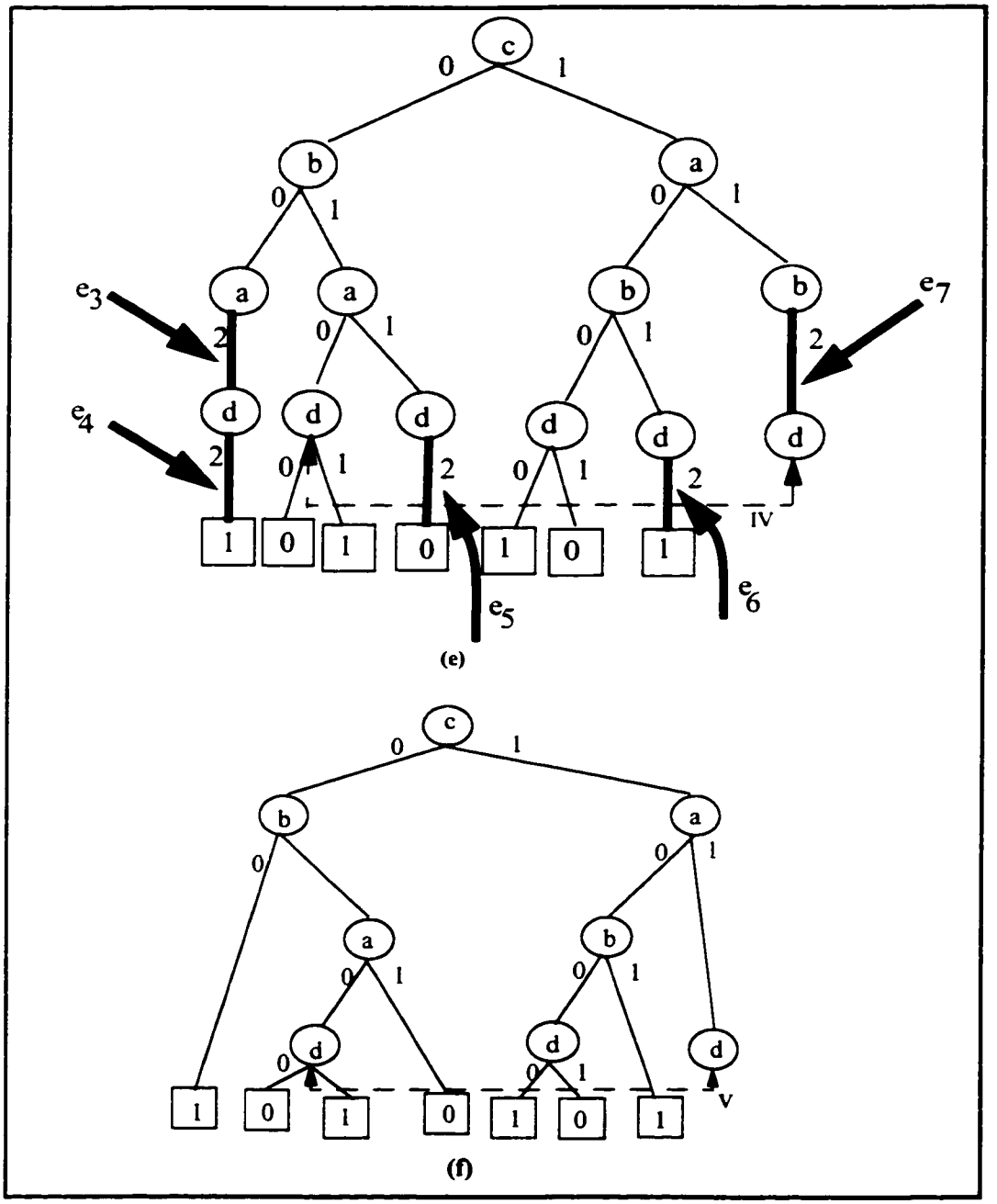

Figure 3.14 : Example of logic optimization

3.6 Technology Mapping Algorithm

In this phase, the resultant BDD, obtained through logic optimization is mapped into combination logic blocks (CLBs). Any of the standard mapping algorithms, for mapping to ACT1 and ACT2 cells may be used. For our experiments we have used a simplified version of the ACT1 cell, where one of the inputs of the OR gate is tied to zero. This simplified cell, called ACT cell, is shown below.

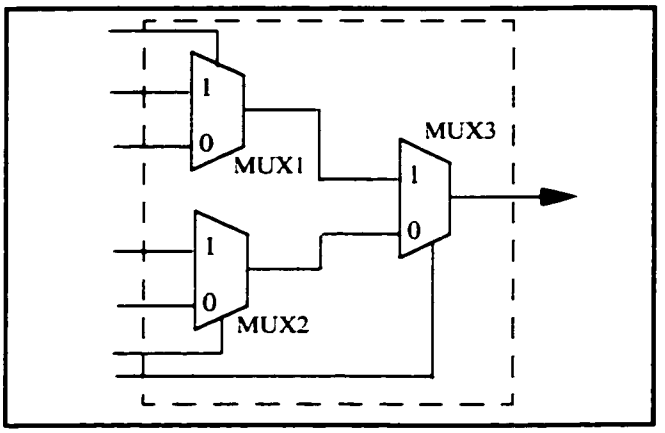

Figure 3.15 : Diagram of ACT cell

We have used a straightforward procedure for mapping the finally optimized unordered BDD onto an interconnection of ACT cells. To determine the position of each new ACT cell, we use the following rules:

1. The root node (representing the primary output) corresponds to the output of one new ACT cell.

2. For any nonterminal node in the network, a new ACT cell is required if

a) it has been merged with other node(s) and has been expanded to the next level

or

b) it is two levels below a node which corresponds to a new ACT cell and at least one of its outgoing branches is connected to a nonterminal node.

3. A terminal node does not require a new ACT cell

4. A nonterminal node with both outgoing branches connected to terminal nodes does not require a new ACT cell.

The total number of cells required to implement a given function and their respective positions is obtained by traversing the entire decision diagram. At each node the above rules are used to determine whether or not a new cell is required at that position.

3.6.1 An Example of Technology Mapping

As an example, let us consider the function $f = acd + \bar{a}bd + ac\bar{d} + b\bar{c}\bar{d}$. We are going to demonstrate our technique using two different representations of the function, one based on ROBDD and the other on 123DD based synthesis. The initial variable order for this function is *c, a, d, b.* The ROBDD representation of the function is shown in Figure 3.16 (a) and the number of ACT cells (three) corresponding to this representation is shown in Figure 3.16 (b).

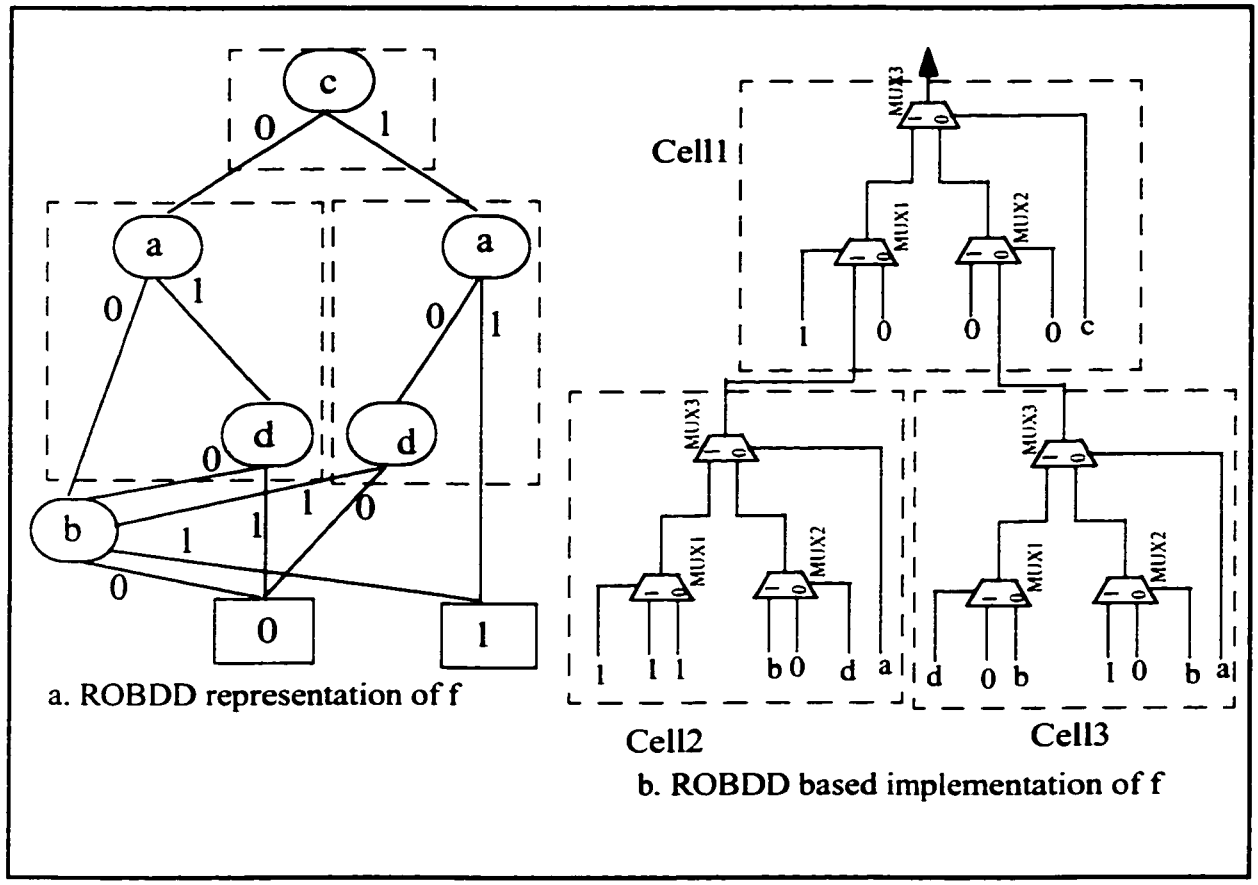

Figure 3.16 Representation and implementation of f with the ROBDD approach

For the 123DD approach, we first construct the 123DD for function f using the same variable ordering. The resultant 123DD is shown in Figure 3.17 (a). After identifying the branches with edges labelled 2 and changing the order of the variable in the corresponding branches, we obtain the final BDD (reduced 123DD) shown in Figure 3.17 (b). The number of ACT cells needed to implement this function is one. The mapping onto the ACT cell is shown in Figure 3.17 (c).

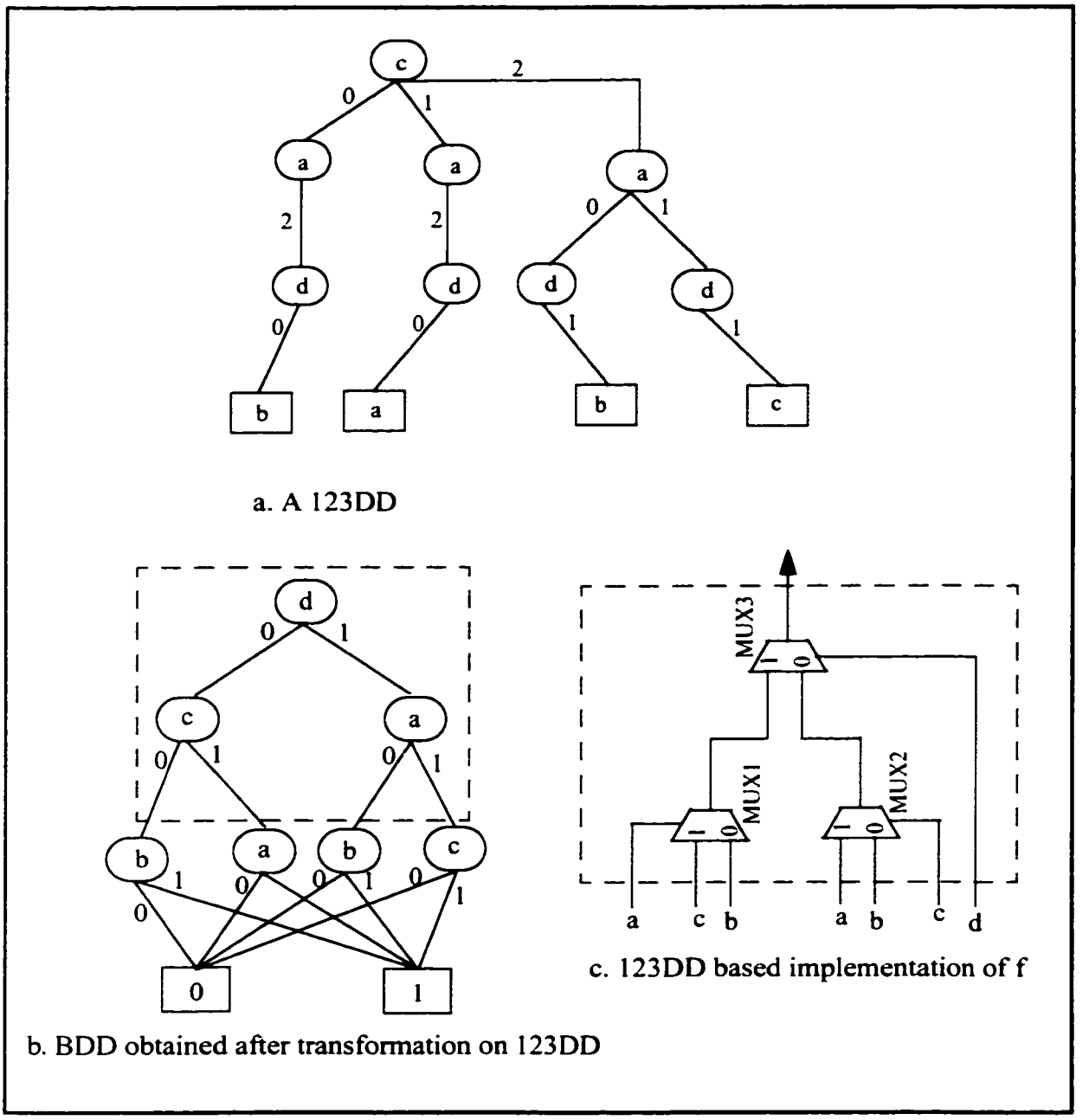

Figure 3.17 Representation and implementation of f with 123DD approach

3.7 Conclusion

In this chapter we have introduced and discussed a new model, the 123DD, for representing Boolean functions. This model is based on the concept of decision diagrams.

The rules for synthesizing the 123DD and performing the transformations on it were described in detail. Finally we showed how such an optimized representation can be mapped to an FPGA architecture consisting of multiplexor-based ACT cells.

Chapter 4

Experimental Results

4.1 Introduction

In this chapter, we discuss the results that were obtained from our experiments using the synthesis procedures introduced in chapter 3. Our synthesis procedure is for single output functions. So in order to successfully test the approach on benchmark circuits, we needed to obtain a network of single output function that will correspond to the benchmark circuits. To do this we used the *Boolean Network* approach during the testing of our synthesis procedure on the benchmark circuits. The concept of Boolean Networks is explained in the next section.

4.2 Boolean Network

A Boolean network is a technology independent multi-level logic structure which can be used to represent multi-output Boolean functions. A large multi-output function is decomposed into a number of smaller functions, which are then interconnected to

realize the original function. We have used a multi-level logic optimization system, MIS [13], to synthesize the Boolean network. A Boolean network consists of a set of interconnected nodes, each representing a smaller single output function. The area complexity of the network is the sum over all nodes of the area complexity of each node.

4.3 The Experimental results

Table 1 shows the cost comparison for a number of MCNC benchmark circuits synthesized using the 123DD approach and the traditional ROBDD based techniques.

Table 1: Area Comparisons for Benchmark Circuits

Function	Number of Subfunctions	Number of ACT cells required for		Percentage Improvement
		123DD	ROBDD	
5v_fns	14	40	44	10.00
6v_fns	14	43	71	65.12
7v_fns	11	68	75	10.29
8v_fns	12	92	103	11.96
apex7	49	166	188	13.25
b9	22	67	74	10.45
C17	2	4	4	0.00
C499	36	36	36	0.00
cc	12	44	46	4.55
cht	36	54	54	0.00
cm151a	8	8	8	0.00
cm152a	2	110	190	73.73
cm162a	8	25	25	0.00
cm163a	5	18	24	33.33

Table 1: Area Comparisons for Benchmark Circuits

Function	Number of Subfunctions	Number of ACT cells required for		Percentage Improvement
		123DD	ROBDD	
cm42a	10	20	20	0.00
cm82a	3	9	11	22.22
cm85a	3	22	23	4.55
cmb	14	27	27	0.00
cnt2	17	140	196	40.00
alu2	6	119	128	7.56
comp	11	93	97	4.30
count	4	48	52	8.33
cu	14	37	40	8.11
decod	16	32	32	0.00
dsip	10	42	50	19.05
f51m	7	39	41	5.13
i1	14	20	20	0.00
majority	1	3	3	0.00
parity	3	17	17	0.00
pcle	16	35	39	11.43
pcler8	20	42	44	4.76
pm1	10	33	36	9.09
sbc	23	53	58	9.43
sct	16	56	63	12.50
tcon	16	8	8	0.00
unreg	16	91	110	20.88
z4ml	4	25	25	0.00

4.3.1 Discussion of Results

In the above experiment, we considered 37 Benchmark functions. The overall improvement for all the functions was 11.35%. This includes 13 functions for which both approaches gave the same results. If we consider only the 24 functions which showed a positive improvement, the average improvement is 17.5%. It is important to note that in all cases, the results obtained using the 123DD approach was better than or equal to results obtained using ROBDD approach. There may be pathological cases where this is not the case. But such cases will be extremely rare and we have not encountered any in our experiments.

Analyzing the results given in Table 1 , we see that for some of the functions (e.g. cm42a, cmb, z4ml) there is little or no improvement, whereas other functions (e.g. cnt2, cm152a, am163a) are greatly improved by using 123DD. It is therefore important to try to identify the functions for which our approach is likely be most beneficial. Given the structure of a general ACT cell, we can see that any single-output function of two or three variables can always be implemented by a single ACT cell, irrespective of whether ROBDD or 123DD based methods are being used. For functions where all or most of the constituent subfunctions are of two or three inputs, our method will provide very little improvement. Even for subfunctions of 4 inputs the improvements may be relatively small. Also there are certain classes of circuits, such as decoders, where both approaches always give the same results, irrespective of the size of the subfunction. In general, however, as the size of a subfunction increases, the likelihood of the 123DD model providing significant improvement also increases. So our approach is most effective when a function is

decomposed into relatively large subfunctions.

Table 2 shows the results for the same set of Benchmark circuits when we consider only those subfunctions which have 4 or more inputs. The average improvement for all 34 functions is 12.76%. If we consider only functions with a positive improvement, the average is 18.1%. We note that three of the original Benchmark functions, (C499, cm151a and tcon) had no subfunctions with more than 3 inputs. So, they were not included in Table 2. As expected, we see a modest increase in the overall improvement.

Table 2: Area Comparisons for Circuits with 4 or More Inputs

Function	Number of Subfunctions	Number of ACT cells required for		Percentage Improvement
		123DD	ROBDD	
5v_fns	14	40	44	10.00
6v_fns	14	43	71	65.12
7v_fns	11	68	75	10.29
8v_fns	12	92	103	11.96
apex7	49	155	177	14.19
b9	22	61	68	11.48
C17	2	4	4	0.00
cc	12	43	45	4.65
cht	36	54	54	0.00
cm152a	2	110	190	72.73
cm162a	8	23	23	0.00
cm163a	5	18	24	33.33
cm42a	10	20	20	0.00

Table 2: Area Comparisons for Circuits with 4 or More Inputs

Function	Number of Subfunctions	Number of ACT cells required for		Percentage Improvement
		123DD	ROBDD	
cm82a	3	8	10	25.00
cm85a	3	22	23	4.55
cmb	14	22	22	0.00
cnt2	17	140	196	40.00
alu2	6	117	126	7.69
comp	11	92	96	4.35
count	4	48	52	8.33
cu	14	32	35	9.38
decod	16	32	32	0.00
dsip	10	38	46	21.05
f51m	7	37	39	5.41
i1	14	11	11	0.00
majority	1	3	3	0.00
parity	3	17	17	0.00
pcle	16	30	34	13.33
pcler8	20	32	34	6.25
pm1	10	31	34	9.68
sbc	23	46	51	10.87
sct	16	52	59	13.46
unreg	16	91	110	20.88
z4ml	4	24	24	0.00

4.4 Conclusion

In this chapter we have presented the results of the synthesis experiments based on a number of Benchmark circuits. We compared our approach, based on the 123DD model,

with conventional synthesis techniques based on the ROBDD model. Our methods always performed better than or equal to ROBDD based methods and in many cases provided significant improvements over conventional techniques.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis we have introduced and described a new methodology for automatically implementing Boolean functions on multiplexor-based FPGAs. This technique is based on a new decision diagram based model - the 123DD and has certain advantages over traditional designs. It combines the advantages of ROBDDs and unordered free BDDs. Like the ROBDD based designs, it allows sharing of subgraphs leading to more compact circuits. In addition, it allows flexibility in the variable ordering of different branches. This makes the design less sensitive to input ordering than ROBDD based designs.

We have tested our approach on a large number of Benchmark circuits and compared the results with traditional ROBDD based synthesis techniques. The results indicate that 123DDs can be used to significantly improve the logic optimization phase in FPGA

design. The results also indicate that the amount of improvement over traditional methods depends on the size of the individual subfunctions implementing a single multi-output circuit. In general, for a single output function, the average improvement increases with the size of the function.

5.2 Future Work

This thesis has shown that the results of preliminary investigations on the use of 123DDs to synthesize multiplexor-based FPGAs look very promising. In this work, we have used the generalized ACT cell as the target architecture for technology mapping. One possible extension of this work would be to use the ACT1 and ACT2 cells the target architecture. The logic optimization phase of the synthesis procedure would remain the same. Only the technology mapping phase needs to be modified to accommodate different target architectures.

In this thesis we have mainly focussed on reducing the area occupied by the circuit, by reducing the number of cells required to implement it. We have not considered the cost of the routing resources needed to interconnect these cells. Further work can be done in this area which takes into account the costs associated with routing as well as the area occupied by the cells.

The main goal of this thesis was to design area-efficient circuits. In some cases however, it may be necessary to ensure that the circuit can operate at or over certain minimum speeds. Under these circumstances, we may need to sacrifice area in order to reduce delays along critical paths. It would be very useful to enhance our synthesis procedure so that it can

take into account additional factors such as speed. Other parameters such as power consumption and noise immunity can also be incorporated into the design.

References

[1] Dinesh Bhatta "Field Programmable Gate Arrays A cheaper way of customizing product prototypes". IEEE Potentials, Feb. 1994, Vol. 13 No. 1 pp 16 - 19

[2] Stephen Brown, Jonathan Rose, "FPGA and CPLD Architectures: A Tutorial" IEEE Design and Test of Computers, vol. 13, No. 2 pp. 42-57

[3] Stephen Brown, Robert J. Francis, Jonathan Rose, Zvonko G. Vranesic "Field-Programmable Gate Arrays", Kluwer Academic Publishers, Norwell, Mass., 1992

[4] Stephen Brown, "FPGA Architectural research: A Survey", IEEE Design & Test of Computers, Winter 1996, vol. 13 No. 4, pp 9-15

[5] Sheldon B. Akers, "Binary Decision Diagrams", IEEE Transactions on Computers, vol. C-27, No. 6, June 1978

[6] C. Y. Lee, Representation of switching circuits by binary-decision programs, Bell Syst. Tech. J., 38, 985-999,1995.

[7] B. M. E. Moret, Decision trees and diagrams, Computing Surveys 14, 593-623, 1982.

[8] Rajeev Murgai, Alberto Sangiovanni-Vincentelli, "Logic Synthesis for Field-Programmable gate Arrays", Kluwer Academic Publishers, Boston, 1995

[9] Arunita Jaekel, "Synthesis Of Multilevel Pass Transistor Logic Networks", PhD Thesis, University of Windsor, 1995.

[10] T. Besson, H. Bouzouzou, M. Crastes, I. Floricica, G. Saucier, "Synthesis on Multiplexer-based FPGA using Binary Decision Diagram", IEEE Proc. of Int. conf. on Computer Design 1992, pp. 163-167

[11] Rajeev Murgai, Yoshihito Nishizaki, Narendra Shenov, Robert K. Brayton, Alberto Sangiovanni-Vincentelli, "Logic Synthesis for Field-Programmable gate Arrays" 27th ACM/IEEE Design Automation Conference, 1990, pp. 620-625

[12] Rajeev Murgai, Robert K. Brayton, Alberto Sangiovanni-Vincentelli, "An Improved Synthesis for Multiplexor-based PGAs", 29th ACM/IEEE Design Automation Conference, 1992, pp. 380-386

[13] Robert K. Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, Albert R. Wang, "MIS: A Multiple-Level Logic Optimization System", IEEE Transaction on Computer-Aided Design, vol., CAD-6, No. 6, November 1987

[14] Hiroshi Sawada, Takayuki Suyama, Akira Nagoya, "Ligic Synthesis for Look-Up Table based FPGAs using Functional Decomposition and Support Minimization", IEEE, 1995.

[15] Jesse H. Jenkins, "Designing with FPGAs and CPLDs", PTR Prentice Hall, 1994.

[16] Pak K. Chan, Samiha Mourad, "Digital Design Using Field Programmable Gate Arrays", Prentice Hall, 1994.

[17] Stephen M. Trimberger, " Field-Programmable Gate Array Technology", Kluwer Academic Publishers, 1994.

[18] A.Armah, A. Jaekel, "An Ordering-Insensitive Methodology for Efficient DCVS Circuit Synthesis", IEEE Canadian Conference on Electrical and Computer Engineering, Waterloo, Ontario, Canada, May 24-28, 1998

[19] Anthony Armah and Arunita Jaekel,"Synthesis of Multiplexor-Based FPGA Using 123-Decision Diagrams", IEEE Instrumentation and measurement Technology Conference, St. Paul, Minnesota, USA, May 18-21, 1998

[20] Kurt Keutzer, "DAGON: Technology Binding and Local Optimization by DAG Matching", 24th ACM/IEEE Design Automation Conference, Paper 21.1, pp 341-347

[21] K. Karplus, "Amap: A Technology Mapper for Selector-based Field-Programmable Gate Arrays", Proc. 28th Design Automation Conference, June 1991

[22] S. Ercolani and G. D. Micheli, "Technology Mapping for Electrically Programmable Gate Arrays ", Proc. 28th Design Automation Conference, June 1991.

[23] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton and A. Sangiovanni-Vincentelli, "Logic Synthesis for Programmable Gate Arrays" Proc. 27th Design Automation Conference, June 1990, pp 620-625

[24] E.Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentilli, and A. R. Wang, "Technology Mapping in MIS", Proc. of ICCAD, 1987

[25] K. Karplus, "Xmap: A Technology Mapper for Table-lookup Field-Programmable Gate Arrays", Proc. 28th Design Automation Conference, June 1991, pp. 240-243

Appendix A

The program codes for the mapping onto the ACT cells

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


/* common ancestor up to 'n' levels above the current level */
#define COM_ANC_LEVEL     3  /* set to 1 for obdd */


union bitmaps
  {
    unsigned int bit_str;
    unsigned int *bit_str_ptr;
  };

struct node
  {
```

```
    int         node_id;
    int         var_no;
    int         signal;
    char         flip;
    int         subtree_id;
    int         bits_no;
    int     newcell;
    int     main_brother;
    union bitmaps bitmap1;
    union bitmaps bitmap2;
    struct node   *left_ptr;
    struct node   *right_ptr;
    struct node   *middle_ptr;
    struct node   *brother_ptr;

  };
 FILE *temp_file;



/* copies the contents of the bitmap pointed to by "bitmap1" to the bitmap
   pointed to by "bitmap2" */
void bitmap_copy(int length, unsigned int *bitmap1, unsigned int *bitmap2)
{
   int i;

   if (length <= 32)
```

```
    *bitmap2 = *bitmap1;
  else
    for (i = 0; i < length/32; i++)
      *(bitmap2 + i) = *(bitmap1 + i);
}

/* generates bitmaps and bits_no for the left child of the node pointed to by
  "root_ptr" */
void generate_leftchild_bitmaps(struct node *root_ptr, struct node *leftchild)
{
  leftchild -> bits_no = root_ptr -> bits_no / 2;

  /* generate bitmaps */
  if (leftchild -> bits_no == 32)
  {
    (leftchild -> bitmap1).bit_str = *((root_ptr -> bitmap1).bit_str_ptr);
    (leftchild -> bitmap2).bit_str = *((root_ptr -> bitmap2).bit_str_ptr);
  }
  else
    if (leftchild -> bits_no < 32)
    {
        (leftchild -> bitmap1).bit_str =
        ((root_ptr -> bitmap1).bit_str) >> leftchild -> bits_no;
      (leftchild -> bitmap2).bit_str =
      ((root_ptr -> bitmap2).bit_str) >> leftchild -> bits_no;
    }
    else
```

```
    {
        (leftchild -> bitmap1).bit_str_ptr =(root_ptr -> bitmap1).bit_str_ptr;
      (leftchild -> bitmap2).bit_str_ptr =(root_ptr -> bitmap2).bit_str_ptr;
    }
}

/* generates bitmaps and bits_no for the right child of the node pointed to by
  "root_ptr" */
void generate_rightchild_bitmaps(struct node *root_ptr, struct node *rightchild)
{
  rightchild -> bits_no = root_ptr -> bits_no / 2;

  /* generate bitmaps */
  if (rightchild -> bits_no == 32)
  {
    (rightchild -> bitmap1).bit_str = *((root_ptr -> bitmap1).bit_str_ptr+1);
    (rightchild -> bitmap2).bit_str = *((root_ptr -> bitmap2).bit_str_ptr+1);
  }
  else
    if (rightchild -> bits_no < 32)
    {
        (rightchild -> bitmap1).bit_str =
        ((root_ptr -> bitmap1).bit_str) << (32 - rightchild -> bits_no) >>
                                    (32 - rightchild -> bits_no);
      (rightchild -> bitmap2).bit_str =
      ((root_ptr -> bitmap2).bit_str) << (32 - rightchild -> bits_no) >>
                                (32 - rightchild -> bits_no);
```

```
    }
    else
    {
      (rightchild -> bitmap1).bit_str_ptr =
            (root_ptr -> bitmap1).bit_str_ptr + rightchild -> bits_no / 32;
      (rightchild -> bitmap2).bit_str_ptr =
            (root_ptr -> bitmap2).bit_str_ptr + rightchild -> bits_no / 32;
    }
}

/* generates bitmaps for the new node */
void generate_new_bitmaps(struct node *old1_ptr, struct node *old2_ptr,
                  struct node *new_ptr)
{
  int i;

  new_ptr -> bits_no = old1_ptr -> bits_no;
  if (new_ptr -> bits_no <= 32)
  {
    (new_ptr -> bitmap1).bit_str = (old1_ptr -> bitmap1).bit_str;
    (new_ptr -> bitmap2).bit_str = (old1_ptr -> bitmap2).bit_str &
                        (old2_ptr -> bitmap2).bit_str;
  }
  else
  {
    (new_ptr -> bitmap1).bit_str_ptr = malloc(new_ptr -> bits_no / 32 *
                                sizeof(int));
```

```
    (new_ptr -> bitmap2).bit_str_ptr = malloc(new_ptr -> bits_no / 32 *
                                        sizeof(int));
    for (i = 0; i < new_ptr -> bits_no / 32; i++)
    {
      *((new_ptr -> bitmap1).bit_str_ptr + i) =
                    *((old1_ptr -> bitmap1).bit_str_ptr + i);
      *((new_ptr -> bitmap2).bit_str_ptr + i) =
        *((old1_ptr -> bitmap2).bit_str_ptr + i) &
        *((old2_ptr -> bitmap2).bit_str_ptr + i);
    }
  }
}

/* returns :
  0 - if the bitmap has 0's on all bits,
  1 -     "          1's    "     ,
  2*(var_no + 1) + 1 - if the bitmap has n 0's and n 1's,
  2*(var_no + 1)    -        "       n 1's and n 0's,
 -1 - otherwise */
int check_bitmaps(struct node *node_ptr)
{
  unsigned int temp1, temp2;
  int i;

  if (node_ptr -> bits_no <= 32)
  {
    /* all 0's */
```

```
  if ((node_ptr -> bitmap1).bit_str == 0) return 0;
  /* all 1's */
  if ((~((node_ptr -> bitmap1).bit_str | (node_ptr -> bitmap2).bit_str)
        << 32 - node_ptr -> bits_no) == 0)
    return 1;
  /* n 0's and n 1's */
  temp1 = (node_ptr -> bitmap1).bit_str >> node_ptr -> bits_no / 2;
  temp2 = ((node_ptr -> bitmap1).bit_str | (node_ptr -> bitmap2).bit_str)
                    << 32 - node_ptr -> bits_no / 2 >>
                              32 - node_ptr -> bits_no / 2;
  if ((temp1 == 0) && ((~temp2 << 32 - node_ptr -> bits_no / 2) == 0))
    return 2 * (abs(node_ptr -> var_no)) + 1;
  /* n 1's and n 0's */
  temp1 = ((node_ptr -> bitmap1).bit_str | (node_ptr -> bitmap2).bit_str)
                  >> node_ptr -> bits_no/2;
  temp2 = (node_ptr -> bitmap1).bit_str << 32 - node_ptr -> bits_no/2
                          >> 32 - node_ptr -> bits_no/2;
  if (((~temp1 << 32 - node_ptr -> bits_no / 2) == 0) && (temp2 == 0))
    return 2 * (abs(node_ptr -> var_no));
}
else
{
  /* all 0's */
  for (i = 0; (i < node_ptr -> bits_no / 32) &&
        (*((node_ptr -> bitmap1).bit_str_ptr + i) == 0); i++);
  if (i == node_ptr -> bits_no / 32) return 0;
  /* all 1's */
```

```
    for (i = 0; (i < node_ptr -> bits_no / 32) &&
          (~(*((node_ptr -> bitmap1).bit_str_ptr + i)
             | *((node_ptr -> bitmap2).bit_str_ptr + i)) == 0); i++);
    if (i == node_ptr -> bits_no / 32) return 1;
    /* n 0's and n 1's */
    for (i = 0; (i < node_ptr -> bits_no / 64) &&
          (*((node_ptr -> bitmap1).bit_str_ptr + i) == 0) &&
          (~(*((node_ptr -> bitmap1).bit_str_ptr + i +
                    node_ptr -> bits_no / 64)
          | *((node_ptr -> bitmap2).bit_str_ptr + i +
                    node_ptr -> bits_no / 64)) == 0); i++);
    if (i == node_ptr -> bits_no / 64)
      return 2 * (abs(node_ptr -> var_no)) + 1;
    /* n 1's and n 0's */
    for (i = 0; (i < node_ptr -> bits_no / 64) &&
          (~(*((node_ptr -> bitmap1).bit_str_ptr + i) |
            *((node_ptr -> bitmap2).bit_str_ptr + i)) == 0) &&
          (*((node_ptr -> bitmap1).bit_str_ptr + i +
                    node_ptr -> bits_no / 64) == 0); i++);
    if (i == node_ptr -> bits_no / 64)
      return 2 * (abs(node_ptr -> var_no));
  }
  return -1;
}

/* returns :
  1 if bitmaps of two nodes : pointed to by "node1_ptr" and "node2_ptr" are
```

```
    equivalent,
  0 otherwise */
int are_equivalent_bitmaps(struct node *node1_ptr, struct node *node2_ptr)
{
  int i;

  if (node1_ptr -> bits_no <= 32)
    if (((node1_ptr -> bitmap1).bit_str & ~(node2_ptr -> bitmap2).bit_str) ==
       ((node2_ptr -> bitmap1).bit_str & ~(node1_ptr -> bitmap2).bit_str))
      return 1;
    else
      return 0;
  else
  {
    for (i = 0; (i < node1_ptr -> bits_no) &&
      ((*((node1_ptr -> bitmap1).bit_str_ptr + i) &
       ~(*((node2_ptr -> bitmap2).bit_str_ptr + i))) ==
       (*((node2_ptr -> bitmap1).bit_str_ptr + i) &
       ~(*((node1_ptr -> bitmap2).bit_str_ptr + i)))); i++);
    if (i == node1_ptr -> bits_no / 32)
      return 1;
    else
      return 0;
  }
}

/* compares bitmaps of two nodes;
```

```
  returns 1 if they are equivalent,
          -1 if the first bitmap has 0's and the second has 1's on all bits,
          -2 if the first bitmap has 1's and the second has 0's on all bits,
           0 otherwise */

int bitmap_compare(struct node *node1_ptr, struct node *node2_ptr)
{
  int i;

  if ((node1_ptr -> signal == 0) && (node2_ptr -> signal == 1))
    return -1;
  if ((node1_ptr -> signal == 1) && (node2_ptr -> signal == 0))
    return -2;
  if (are_equivalent_bitmaps(node1_ptr, node2_ptr) == 1)
    return 1;

  return 0;
}

/* creates children of the node pointed to by "root_ptr" and returns a
  pointer to the left child */
struct node *create_children(struct node *root_ptr, int level_no)
{
  struct node *leftchild, *rightchild;

  leftchild = malloc(sizeof(struct node));
  if (leftchild == NULL)
```

```
{
  printf("malloc failed\n");
  return NULL;
}
rightchild = malloc(sizeof(struct node));
if (rightchild == NULL)
{
  printf("malloc failed\n");
  return NULL;
}

leftchild -> node_id = 3 * (root_ptr -> node_id);
rightchild -> node_id = leftchild -> node_id + 1;

leftchild -> var_no = - level_no;
rightchild -> var_no = level_no;

leftchild -> flip = 'n';
rightchild -> flip = 'n';

leftchild -> newcell = 0;
rightchild -> newcell = 0;

leftchild -> main_brother = 0;
rightchild -> main_brother = 0;

leftchild -> subtree_id = root_ptr -> subtree_id;
```

```
  rightchild -> subtree_id = root_ptr -> subtree_id;

  /* generate bitmaps */
  generate_leftchild_bitmaps(root_ptr, leftchild);
  generate_rightchild_bitmaps(root_ptr, rightchild);

  /* generate signal variables */
  leftchild -> signal = check_bitmaps(leftchild);
  rightchild -> signal = check_bitmaps(rightchild);

  leftchild -> left_ptr = NULL;
  leftchild -> right_ptr = NULL;
  leftchild -> middle_ptr = rightchild;
  rightchild -> left_ptr = NULL;
  rightchild -> right_ptr = NULL;
  rightchild -> middle_ptr = NULL;
  leftchild -> brother_ptr = NULL;
  rightchild -> brother_ptr = NULL;

  root_ptr -> left_ptr = leftchild;
  root_ptr -> right_ptr = rightchild;

  return leftchild;
}

/* expands tree by one level; returns a pointer to the most-left node in this
  level*/
```

```
struct node *expand_tree(struct node *root_ptr, int level_no,
                         int *fixed_sign_no)
{
  struct node *children_ptr;

  if (root_ptr == NULL) return NULL;

  if (root_ptr -> signal < 2)
  {
    children_ptr = create_children(root_ptr, level_no);
    (root_ptr -> right_ptr) -> middle_ptr = expand_tree(
                                        root_ptr -> middle_ptr,
                                        level_no, fixed_sign_no);
  }
  else
  {
    if (root_ptr -> signal > 1)
      (*fixed_sign_no)++;
    /* do not expand nodes with fixed signal variable */
    children_ptr = expand_tree(root_ptr -> middle_ptr,level_no,fixed_sign_no);
  }

  /* after children have been created, unlink the list of parents */
  root_ptr -> middle_ptr = NULL;

  return children_ptr;
}
```

```
/* returns the number of nodes in the level */
int count_nodes(struct node *level_ptr)
{
  int count = 0;

  while (level_ptr != NULL)
  {
    count++;
    level_ptr = level_ptr -> middle_ptr;
  }
  return count;
}

/* returns :
  0  - if the node containing "node_id" is the left child ,
  1  -      "            "          right child,
  2  -      "            "          middle child
  of the node pointed to by "parent_ptr",
  returns -1 otherwise */
int determine_which_child(struct node *parent_ptr, int node_id)
{
  if ((parent_ptr -> left_ptr != NULL) &&
     ((parent_ptr -> left_ptr) -> node_id == node_id)) return 0;
  else
  if ((parent_ptr -> right_ptr != NULL) &&
     ((parent_ptr -> right_ptr) -> node_id == node_id)) return 1;
```

```
  else
  if ((parent_ptr -> middle_ptr != NULL) &&
     ((parent_ptr -> middle_ptr) -> node_id == node_id)) return 2;
  else return -1;
}

/* searches for a node with node_id equal "node_id" in the tree pointed to by
  "root_ptr" and returns a pointer to this node, or NULL if not found */
struct node *find_node(int node_id, struct node *root_ptr)
{
  int i;

  if (root_ptr -> node_id == node_id) return root_ptr;

  for (i = 3; (node_id / i) != root_ptr -> node_id; i *= 3);
  switch (determine_which_child(root_ptr, node_id / (i / 3)))
  {
    case  0 : return find_node(node_id, root_ptr -> left_ptr);
    case  1 : return find_node(node_id, root_ptr -> right_ptr);
    case  2 : return find_node(node_id, root_ptr -> middle_ptr);
    default : return NULL;
  }
}

/* returns a pointer to the common ancestor of two nodes */
struct node *find_common_ancestor(int node1_id, int node2_id,
                                  struct node *root_ptr)
```

```
{
  int i;

  for (i = 3; node1_id / i != node2_id / i; i *= 3);
  return find_node(node1_id / i, root_ptr);
}

/* checks if nodes : pointed to by "node_ptr" and pointed to by "sibl_ptr" are
  mergable cousins; returns :
  0 if they are not mergable,
  1, -1, -2, or -3 if they are mergable
  1 - means that they have identical bitmaps
  -1 - the first has all 0's, the second has all 1's
  -2 - the first has all 1's, the second has all 0's
  -3 - they have identical signal variables != -1 */
int is_mergable_cousin(struct node *node_ptr, struct node *sibl_ptr,
                       struct node *main_root_ptr)
{
  int stat, i, j;
  struct node *temp1_ptr, *temp2_ptr, *common_anc_ptr;

  if (node_ptr -> subtree_id != sibl_ptr -> subtree_id) return 0;

  /* check variable numbers */
  if (node_ptr -> var_no != sibl_ptr -> var_no) return 0;

  /* check signal variables */
```

```
if ((node_ptr -> signal != sibl_ptr -> signal) &&
   ((node_ptr -> signal != 1) || (sibl_ptr -> signal != 0)) &&
   ((node_ptr -> signal != 0) || (sibl_ptr -> signal != 1)))
   return 0;

if ((node_ptr -> signal != sibl_ptr -> signal) || (node_ptr -> signal == -1))
{
  /* check bitmaps */
  stat = bitmap_compare(node_ptr, sibl_ptr);

  if (stat == 0) return 0;
}
/* check if cousins */
common_anc_ptr = find_common_ancestor(node_ptr -> node_id,
                                      sibl_ptr -> node_id,
                                      main_root_ptr);
j = 1;
for (i = 3; node_ptr -> node_id / i != common_anc_ptr -> node_id; i *= 3)
  j++;

/* check above */
if (j > COM_ANC_LEVEL)
  return 0;

/* follow two paths checking variable numbers */
temp1_ptr = find_node(node_ptr -> node_id / (i / 3), common_anc_ptr);
temp2_ptr = find_node(sibl_ptr -> node_id / (i / 3), common_anc_ptr);
```

```
  if (temp1_ptr -> var_no != - temp2_ptr -> var_no) return 0;

  for (j = 9; j < i; j *= 3)
  {
    temp1_ptr = find_node(node_ptr -> node_id / (i / j), temp1_ptr);
    temp2_ptr = find_node(sibl_ptr -> node_id / (i / j), temp2_ptr);
    if (temp1_ptr -> var_no != temp2_ptr -> var_no) return 0;
  }

  /* the two nodes are cousins */
  if ((node_ptr -> signal == sibl_ptr -> signal) && (node_ptr -> signal != -1))
    return -3;
  else
    return stat;
}

/* deletes a node containing "node_id" from the list pointed to by "list_ptr";
  returns a pointer to the list */
struct node *list_delete(int node_id, struct node *list_ptr)
{
  /* list is not empty; the node to be deleted is in the list */

  if (list_ptr -> node_id == node_id)
    return list_ptr -> middle_ptr;

  list_ptr -> middle_ptr = list_delete(node_id, list_ptr -> middle_ptr);
  return list_ptr;
```

```
}

/* returns a pointer to the last element of the list */
struct node *find_end_of_list(struct node *list_ptr)
{
  /* list can't be  empty */

  if (list_ptr -> middle_ptr == NULL) return list_ptr;

  return find_end_of_list(list_ptr -> middle_ptr);
}


/* deletes the node containing "node_id" from the tree pointed to by "root_ptr"
  and recursively deletes all ancestors of this node with no children */
void tree_node_delete(int node_id, struct node *root_ptr)
{
  struct node *parent_ptr;

  parent_ptr = find_node(node_id / 3, root_ptr);

  switch (determine_which_child(parent_ptr, node_id))
  {
    case 0 : parent_ptr -> left_ptr = NULL;
          break;
    case 1 : parent_ptr -> right_ptr = NULL;
          break;
```

```
    case 2 : parent_ptr -> middle_ptr = NULL;
             break;
    default: break;
  }
  if ((parent_ptr -> left_ptr == NULL) && (parent_ptr -> right_ptr == NULL) &&
     (parent_ptr -> middle_ptr == NULL))
    tree_node_delete(parent_ptr -> node_id, root_ptr);
}

/* prints all nodes of the level pointed to by "level_ptr" */
void print_level(struct node *level_ptr)
{
  struct node *temp_ptr = level_ptr;
  int i;

  printf("LEVEL : \n");
  while (temp_ptr != NULL)
  {
    printf("node_id: %d, var_no: %d, bits_no: %d, signal: %d\n",
                                temp_ptr -> node_id,
                                        temp_ptr -> var_no,
                                temp_ptr -> bits_no,
                                temp_ptr -> signal);
    if (temp_ptr -> bits_no <= 32)
        printf("%08x %08x\n", (temp_ptr -> bitmap1).bit_str,
                        (temp_ptr -> bitmap2).bit_str);
    else
```

```
    {
        for (i = 0; i < temp_ptr -> bits_no / 32; i++)
          printf("%08x", *((temp_ptr -> bitmap1).bit_str_ptr + i));
        printf("\n");
      for (i = 0; i < temp_ptr -> bits_no / 32; i++)
        printf("%08x", *((temp_ptr -> bitmap2).bit_str_ptr + i));
      printf("\n");
    }
    temp_ptr = temp_ptr -> middle_ptr;
  }
}

/* creates a node with id equal "node_id" and variable number equal "var_no";
  returns a pointer to this node */
struct node *create_path_node(int node_id,
                              int var_no)
{
  struct node *new_node_ptr;

  new_node_ptr = malloc(sizeof(struct node));
  if (new_node_ptr == NULL)
  {
    printf("malloc failed\n");
    exit(EXIT_FAILURE);
  }
  new_node_ptr -> node_id = node_id;
  new_node_ptr -> var_no = var_no;
```

```
  new_node_ptr -> signal = -1;
  new_node_ptr -> flip = 'n';
  new_node_ptr -> bits_no = 0;
  new_node_ptr -> subtree_id = 1;
  new_node_ptr -> newcell = 0;
  new_node_ptr -> main_brother = 0;
  new_node_ptr -> left_ptr = NULL;
  new_node_ptr -> right_ptr = NULL;
  new_node_ptr -> middle_ptr = NULL;
  new_node_ptr -> brother_ptr = NULL;

  return new_node_ptr;
}

/* creates a new path pointed to by "new_parent" corresponding to the path
  pointed to by "old_parent"; returns a pointer to the last node in the new
  path */
struct node *create_new_path(struct node *new_parent,
                             struct node *old_parent,
                    int old_leaf_id,
                             int counter)
{
  struct node *old_node_ptr, *temp_ptr;
  int dir, which_child;

  /* old_node_ptr points to the node to be "copied" */
  old_node_ptr = find_node(old_leaf_id / counter, old_parent);
```

```
switch (old_node_ptr -> node_id % 3)
{
  /* if corresponding node in the new path does not exists,
     create a new node */
  case  0 : if ((dir = determine_which_child(new_parent,
                      3*new_parent -> node_id)) == -1)
            temp_ptr = create_path_node(
                      3 * new_parent -> node_id,
                      old_node_ptr -> var_no);
          else
            temp_ptr = find_node(3*new_parent -> node_id, new_parent);
              which_child = 0;
          break;
  case  1 : if ((dir = determine_which_child(new_parent,
                      3*new_parent -> node_id + 1)) == -1)
            temp_ptr = create_path_node(
                      3 * new_parent -> node_id + 1,
                      old_node_ptr -> var_no);
          else
            temp_ptr = find_node(3*new_parent -> node_id+1, new_parent);
              which_child = 1;
          break;
  case  2 : if ((dir = determine_which_child(new_parent,
                      3*new_parent -> node_id + 2)) == -1)
            temp_ptr = create_path_node(
                      3 * new_parent -> node_id + 2,
                      old_node_ptr -> var_no);
```

```
            else
              temp_ptr = find_node(3*new_parent -> node_id+2, new_parent);
                 which_child = 2;
            break;
    default : break;
  }
  if (dir == -1)
    if ((new_parent -> left_ptr == NULL)&&(which_child == 0))
      new_parent -> left_ptr = temp_ptr;
    else
    if ((new_parent -> right_ptr == NULL)&&(which_child == 1))
      new_parent -> right_ptr = temp_ptr;
    else
      new_parent -> middle_ptr = temp_ptr;

  /* stop, if the whole path has been copied */
  if (old_node_ptr -> node_id == old_leaf_id)
    return temp_ptr;
  else
    return create_new_path(temp_ptr, old_node_ptr,
                    old_leaf_id, counter / 3);
}

/* updates values of the signal variable and bitmasks in the new_leaf */
void new_leaf_node_update(struct node *new_leaf_ptr,
                          struct node *cousin1_ptr, struct node *cousin2_ptr,
                          struct node *common_anc_ptr, int stat)
```

```
{
  int i, var_no;

  if (common_anc_ptr -> left_ptr != NULL)
    var_no = abs((common_anc_ptr -> left_ptr) -> var_no);
  else
    var_no = abs((common_anc_ptr -> right_ptr) -> var_no);
  generate_new_bitmaps(cousin1_ptr, cousin2_ptr, new_leaf_ptr);
  new_leaf_ptr -> subtree_id = cousin1_ptr -> subtree_id;
  switch (stat)
  {
    case -1 :/* cousin1 has 0's on all bits -- the signal variable of the
                new node depends on whether or not cousin1 is in the left
                subtree of the common ancestor */
              for (i = 3; cousin1_ptr -> node_id / i !=
                   common_anc_ptr -> node_id; i *= 3);
              if ((cousin1_ptr -> node_id / (i / 3)) % 3 == 0)
                    new_leaf_ptr -> signal = 2 * var_no + 1;
                  else
                    new_leaf_ptr -> signal = 2 * var_no;
              break;
    case -2 : /* cousin2 has 0's on all bits */
              /* similar to the case -1 */
              for (i = 3; cousin2_ptr -> node_id / i !=
                        common_anc_ptr -> node_id; i *= 3);
              if ((cousin2_ptr -> node_id / (i / 3)) % 3 == 0)
                    new_leaf_ptr -> signal = 2 * var_no + 1;
```

```
                else
                  new_leaf_ptr -> signal = 2 * var_no;
                break;
    case -3 : /* both cousins have identical signal variables != -1 */
            new_leaf_ptr -> signal = cousin1_ptr -> signal;
                break;
    default : break;
  }
}

/* merges two cousins : pointed to by "cousin1_ptr" and pointed to by
  "cousin2_ptr" and returns a pointer to the new node */
struct node *merge_two_cousins(
                              struct node *cousin1_ptr,
                              struct node *cousin2_ptr,
                              struct node *root_ptr,
                              int stat)
{
  struct node *new_leaf_ptr, *common_anc_ptr, *temp_ptr;
  int i, dir;

  for (i = 9; cousin1_ptr -> node_id / i != cousin2_ptr -> node_id / i;
     i *= 3);
  common_anc_ptr = find_node(cousin1_ptr -> node_id / i, root_ptr);
  /* if the phi node does not exist, create it */
  if (determine_which_child(common_anc_ptr, 3*common_anc_ptr -> node_id + 2)
     == -1)
```

```
  {
    /* create phi node */
abs(commo[illegible] = create_path_node(3 * common_anc_ptr -> node_id + 2,
    if (common_anc_ptr -> left_ptr == NULL)
      common_anc_ptr -> left_ptr = temp_ptr;
    else
    if (common_anc_ptr -> right_ptr == NULL)
      common_anc_ptr -> right_ptr = temp_ptr;
    else
      common_anc_ptr -> middle_ptr = temp_ptr;
  }
  /* create a new path that starts at the phi node */
  new_leaf_ptr = create_new_path(
            find_node(3*common_anc_ptr -> node_id + 2, common_anc_ptr),
            find_node(cousin1_ptr -> node_id / (i / 3), common_anc_ptr),
                 cousin1_ptr -> node_id, i / 9);

  /* update signal variable of the new leaf */
  new_leaf_node_update(new_leaf_ptr, cousin1_ptr, cousin2_ptr,
                common_anc_ptr, stat);

  /* delete both cousins and all their ancestors with no children from the
    tree */
  tree_node_delete(cousin1_ptr -> node_id, common_anc_ptr);
  tree_node_delete(cousin2_ptr -> node_id, common_anc_ptr);

  return new_leaf_ptr;
```

```
}

/* searches the level pointed to by "level_ptr" for mergable cousins and merges
   them; returns a pointer to the level */
struct node *merge_cousins(struct node *level_ptr, struct node *root_ptr)
{
  struct node *cousin1_ptr, *cousin2_ptr, *level_last_ptr, *new_leaf_ptr,
              *next_ptr;
  int stat, found = 0;

  level_last_ptr = find_end_of_list(level_ptr);

  cousin1_ptr = level_ptr;
  while (cousin1_ptr != NULL)
  {
    cousin2_ptr = cousin1_ptr -> middle_ptr;
    while ((cousin2_ptr != NULL) && (found == 0))
    {
        if ((stat = is_mergable_cousin(cousin1_ptr, cousin2_ptr, root_ptr))
            != 0)
        {
       /* are mergable */
/*1*/     /* printf("%d, %d, ", cousin1_ptr -> node_id, cousin2_ptr -> node_id);*/
          new_leaf_ptr = merge_two_cousins(cousin1_ptr, cousin2_ptr,
                                           root_ptr, stat);
/*2*/     /*printf("node_id = %d, var_no = %d, signal = %d\n",
          new_leaf_ptr -> node_id, new_leaf_ptr -> var_no,
```

```
            new_leaf_ptr -> signal);*/
        /* insert the new node at the end of the list */
            level_last_ptr -> middle_ptr = new_leaf_ptr;
            level_last_ptr = new_leaf_ptr;
        /* determine the next node to check */
        next_ptr = cousin1_ptr -> middle_ptr;
        if (next_ptr -> node_id == cousin2_ptr -> node_id)
          next_ptr = next_ptr -> middle_ptr;
        /* delete merged nodes from the list */
            level_ptr = list_delete(cousin1_ptr -> node_id, level_ptr);
        free(cousin1_ptr);
            level_ptr = list_delete(cousin2_ptr -> node_id, level_ptr);
        free(cousin2_ptr);
            found = 1;
         }
         else
            cousin2_ptr = cousin2_ptr -> middle_ptr;
    }
    if (found == 1)
    {
      cousin1_ptr = next_ptr;
      found = 0;
    }
    else
      cousin1_ptr = cousin1_ptr -> middle_ptr;
}
```

```
  return level_ptr;
}

/* checks if nodes : pointed to by "node_ptr" and pointed to by "sibl_ptr" are
  mergable brothers; returns :
  0 if they are not mergable,
  1 if they are mergable */
int is_mergable_brother(struct node *node_ptr, struct node *sibl_ptr)
{
  /* check if brothers */
  if ((node_ptr -> node_id / 3 == sibl_ptr -> node_id /3) &&
     (abs(node_ptr -> node_id - sibl_ptr -> node_id) == 1))
    /* check bitmaps */
    return are_equivalent_bitmaps(node_ptr, sibl_ptr);
  else
    /* are not brothers */
    return 0;
}

/* merges two brothers : pointed to by "brother1_ptr" and pointed to by
  "brother2_ptr"; returns a pointer to the new node*/
struct node *merge_two_brothers(struct node *brother1_ptr,
                    struct node *brother2_ptr,
                              struct node *root_ptr)
{
  struct node *parent_ptr;
```

```
  /* find the parent and create the phi node */
  parent_ptr = find_node(brother1_ptr -> node_id / 3, root_ptr);
  parent_ptr -> middle_ptr = create_path_node(3* parent_ptr -> node_id + 2,0);
  (parent_ptr -> middle_ptr) -> signal = brother1_ptr -> signal;
  (parent_ptr -> middle_ptr) -> subtree_id = brother1_ptr -> subtree_id;
  generate_new_bitmaps(parent_ptr -> right_ptr, parent_ptr -> left_ptr,
                  parent_ptr -> middle_ptr);
  /* remove brothers from the tree */
  parent_ptr -> right_ptr = NULL;
  parent_ptr -> left_ptr = NULL;

  return parent_ptr -> middle_ptr;
}

/* searches for mergable brothers in the level pointed to by "level_ptr",
  merges them, and returns a pointer to the level */
struct node *merge_brothers(struct node *level_ptr, struct node *root_ptr)
{
  struct node *brother1_ptr, *brother2_ptr, *next_ptr, *level_last_ptr,
          *new_leaf_ptr;
  int stat, found = 0;

  level_last_ptr = find_end_of_list(level_ptr);

  brother1_ptr = level_ptr;
  while (brother1_ptr != NULL)
  {
```

```
brother2_ptr = brother1_ptr -> middle_ptr;
while ((brother2_ptr != NULL) && (found == 0))
{
    if ((stat = is_mergable_brother(brother1_ptr, brother2_ptr))
        == 1)
    {
    /* are mergable */
    found = 1;
      new_leaf_ptr = merge_two_brothers(brother1_ptr, brother2_ptr,
                                root_ptr);

    /* determine the next node to check */
      next_ptr = brother1_ptr -> middle_ptr;
      if (next_ptr -> node_id == brother2_ptr -> node_id)
        next_ptr = next_ptr -> middle_ptr;
    /* insert the new node at the end of the list */
      level_last_ptr -> middle_ptr = new_leaf_ptr;
      level_last_ptr = new_leaf_ptr;
    /* delete brothers from the list */
      level_ptr = list_delete(brother1_ptr -> node_id, level_ptr);
      free(brother1_ptr);
      level_ptr = list_delete(brother2_ptr -> node_id, level_ptr);
      free(brother2_ptr);
      break;
    }
  else
  brother2_ptr = brother2_ptr -> middle_ptr;
```

```
    }
    if (found == 1)
    {
        found = 0;
        brother1_ptr = next_ptr;
    }
    else
      brother1_ptr = brother1_ptr -> middle_ptr;
  }

  return level_ptr;
}



/* searches for the nodes with identical bitmaps in the level pointed to by
  "level_ptr", merges them and returns a pointer to the level */
struct node *merge_nodes_with_identical_bitmaps(struct node *level_ptr,
                                    struct node *root_ptr)
{
  struct node *node1_ptr, *node2_ptr, *next_ptr, *common_anc_ptr;
  int stat, stat1;

  node1_ptr = level_ptr;
  while (node1_ptr != NULL)
  {
    node2_ptr = node1_ptr -> middle_ptr;
```

```
    while (node2_ptr != NULL)
    {
      stat = are_equivalent_bitmaps(node1_ptr, node2_ptr);
        if ((stat == 1) && (node1_ptr -> signal == -1) &&
        (node2_ptr -> signal == -1))
        {

         /* delete the "right" node from the list -- it will not be
           expanded */

         {
             level_ptr = list_delete(node2_ptr -> node_id, level_ptr);
           node2_ptr -> signal = - node1_ptr -> node_id;
           node2_ptr -> brother_ptr = node1_ptr;
           node1_ptr -> subtree_id = node1_ptr -> node_id;
                node1_ptr -> main_brother = 1;
           node2_ptr -> flip = 'a';
           node1_ptr -> flip = 'a';

         }
/*  CHECK LATER*/
         /*node1_ptr = node1_ptr -> middle_ptr;
         if (node1_ptr == NULL) node2_ptr = NULL;
         else node2_ptr = node1_ptr -> middle_ptr;*/


         node2_ptr = node2_ptr -> middle_ptr;
```

```
      }
      else
        node2_ptr = node2_ptr -> middle_ptr;
    }
    if (node1_ptr != NULL)
      node1_ptr = node1_ptr -> middle_ptr;
  }
  return level_ptr;
}

/* creates the tree pointed to by "root_ptr"  */
void create_tree(struct node *level_ptr, struct node *root_ptr, int level_no,
            int *fixed_sig_no, int *max_level_length)
{
  int level_length = *fixed_sig_no;

  if (level_ptr == NULL) return;

/*6*/
  /*print_level(level_ptr);
  printf("merge cousins\n");*/
  level_ptr = merge_cousins(level_ptr, root_ptr);
  /*printf("merge brothers\n");*/
  level_ptr = merge_brothers(level_ptr, root_ptr);
  if (level_ptr -> bits_no > 4)
  {
    /*printf("merge nodes with identical bitmaps\n");*/
```

```
    level_ptr = merge_nodes_with_identical_bitmaps(level_ptr, root_ptr);
  }
  level_length += count_nodes(level_ptr);
  if (level_length > *max_level_length)
    *max_level_length = level_length;

  if (level_ptr -> bits_no <= 2) return;
  else
  {
    level_ptr = expand_tree(level_ptr, level_no, fixed_sig_no);
    create_tree(level_ptr, root_ptr, level_no + 1, fixed_sig_no,
            max_level_length);
  }
}

/* pre_order traversal of the tree pointed to by "root_ptr" */
void traverse_tree(struct node *root_ptr)
{
  int i;

  if (root_ptr == NULL) return;
if (root_ptr -> signal >= -1) /* CHANGED (root_ptr -> signal == -1) TO
(root_ptr -> signal >= -1) */
{
    printf("node %d, left child : ", root_ptr -> node_id);
    if (root_ptr -> left_ptr != NULL)
      printf("%d, right child : ", (root_ptr -> left_ptr) -> node_id);
```

```
    else
      printf("none, right child : ");

    if (root_ptr -> right_ptr != NULL)
      printf("%d, middle child : ", (root_ptr -> right_ptr) -> node_id);
    else
      printf("none, middle child : ");

    if ((root_ptr -> middle_ptr != NULL)&&(root_ptr -> signal == -1))
      printf("%d\n", (root_ptr -> middle_ptr) -> node_id);
    else
      printf("none\n");
    traverse_tree(root_ptr -> left_ptr);
    traverse_tree(root_ptr -> right_ptr);
    if (root_ptr -> signal == -1)
        traverse_tree(root_ptr -> middle_ptr);
  }
  else
    if (root_ptr -> signal < -1)
      printf("node %d, no children, left brother : node %d\n",
           root_ptr -> node_id,  root_ptr -> brother_ptr -> node_id);
if ((root_ptr -> signal > -1)&&(root_ptr -> left_ptr == NULL)&&
        (root_ptr -> right_ptr == NULL))
        printf("node %d, signal %d\n", root_ptr -> node_id, root_ptr -> signal);
}
```

```
int determine_depth(struct node *root_ptr)
{ int stat, stat1, stat2, stat3;

 if (root_ptr == NULL) return 0;
 if (root_ptr -> flip == 'a') return 1;
 if (root_ptr -> left_ptr == NULL) stat1 = 0;
else if ((root_ptr -> left_ptr -> middle_ptr == NULL) &&
         (root_ptr -> left_ptr -> flip != 'a')) stat1 = 0;
else stat1 = 1;

 if (root_ptr -> right_ptr == NULL) stat2 = 0;
else if ((root_ptr -> right_ptr -> middle_ptr == NULL) &&
        (root_ptr -> right_ptr -> flip != 'a')) stat2 = 0;
else stat2 = 1;

 if (root_ptr -> middle_ptr == NULL) stat3 = 0;
else if ((root_ptr -> middle_ptr -> middle_ptr == NULL) &&
        (root_ptr -> middle_ptr -> flip != 'a')) stat3 = 0;
else stat3 = 1;
 stat = stat1 + stat2 + stat3;
  return stat;
}


void remove_null_node(struct node *root_ptr)
{
if ((root_ptr -> left_ptr -> left_ptr == NULL) &&
```

```
        (root_ptr -> left_ptr -> right_ptr == NULL) &&
        (root_ptr -> left_ptr -> middle_ptr == NULL)) root_ptr -> left_ptr = NULL;

if ((root_ptr -> right_ptr -> left_ptr == NULL) &&
        (root_ptr -> right_ptr -> right_ptr == NULL) &&
        (root_ptr -> right_ptr -> middle_ptr == NULL)) root_ptr -> right_ptr = NULL;
}

void switch_level(struct node *root_ptr)
{
  int i, var1_no, var2_no, stat, node_id, var_no;
  struct node *tmp1_ptr, *tmp2_ptr, *tmp_ptr;


     node_id = root_ptr -> node_id ;

    if (root_ptr -> left_ptr == NULL)
      {
      if (root_ptr -> right_ptr != NULL) var_no = -(root_ptr -> right_ptr -> var_no);
      else var_no = -( abs(root_ptr -> var_no) + 1);
      tmp_ptr = create_path_node(3*node_id, var_no);
      root_ptr -> left_ptr = tmp_ptr;


      }
    if (root_ptr -> right_ptr == NULL)
      {
      if (root_ptr -> left_ptr != NULL) var_no = -(root_ptr -> left_ptr -> var_no);
```

```
        else var_no = ( abs(root_ptr -> var_no) + 1);
        tmp_ptr = create_path_node(3*node_id+1, var_no);
        root_ptr -> right_ptr = tmp_ptr;
       }
    tmp1_ptr = root_ptr -> left_ptr -> right_ptr;
    root_ptr -> left_ptr -> right_ptr = root_ptr -> right_ptr -> left_ptr;
    root_ptr -> right_ptr -> left_ptr = tmp1_ptr;

    var1_no = root_ptr -> var_no;
    root_ptr -> var_no = root_ptr -> left_ptr -> var_no;
    root_ptr -> left_ptr -> var_no = var1_no;
    root_ptr -> right_ptr -> var_no = var1_no;
    if (root_ptr -> middle_ptr != NULL)
    {
     root_ptr -> left_ptr -> middle_ptr = root_ptr -> middle_ptr -> left_ptr;
     root_ptr ->right_ptr -> middle_ptr = root_ptr -> middle_ptr -> right_ptr;
     root_ptr -> middle_ptr = NULL;
    }
    remove_null_node(root_ptr);
printf("switched variables %d and %d at node %d \n",abs(var1_no),
        abs(root_ptr -> var_no), root_ptr -> node_id);
   /* printf("sw*printf("TREE :\n");
    traverse_tree(root_ptr);
    printf("\n");*/
}
```

```
void insert_node_if_reqd(struct node *root_ptr, int dir)

{int node_id, var_no;

 if (dir == 0)
   {
    if (root_ptr -> left_ptr != NULL) return;
    var_no = root_ptr -> right_ptr -> var_no;
    node_id = 3*(root_ptr -> node_id) + dir;
    root_ptr -> left_ptr = create_path_node(node_id, var_no);
   }
 if (dir == 1)
   {
    if (root_ptr -> right_ptr != NULL) return;
    var_no = root_ptr -> left_ptr -> var_no;
    node_id = 3*(root_ptr -> node_id) + dir;
    root_ptr -> right_ptr = create_path_node(node_id, var_no);
   }

 return;
} /*end of function */



int find_depth(struct node *root_ptr)
{ int depth = 0;
 struct node *tmp_ptr;
```

```
  if ((root_ptr -> left_ptr != NULL) && (root_ptr -> right_ptr != NULL))
        return 2;

  if (root_ptr -> left_ptr == NULL) tmp_ptr = root_ptr -> right_ptr;
  else tmp_ptr = root_ptr -> left_ptr;

  if ((tmp_ptr -> left_ptr != NULL) && (tmp_ptr -> right_ptr != NULL))
        return 1;
  else if (tmp_ptr -> middle_ptr != NULL) return 1;
        else return 2;


}
struct node * restore_child(struct node *lchild,  int bits, int sep)
{ int node_id, var_no;
 struct node *rchild;

  node_id = lchild -> node_id - sep;
  var_no = lchild -> var_no ;
  rchild = create_path_node(node_id, var_no);
  if (bits > 4)
    {
        rchild -> brother_ptr = lchild;
        rchild -> signal = -(lchild -> node_id);
        lchild -> flip = 'a';
        lchild -> main_brother = 1;
```

```
    }
  else
    {
        rchild -> left_ptr = lchild -> left_ptr;
        rchild -> right_ptr = lchild -> right_ptr;
        rchild -> middle_ptr = lchild -> middle_ptr;
        rchild -> signal = lchild -> signal;
    }
return rchild;

}




void restore_branch(struct node *root_ptr)

{int i=0, node_id, var_no, depth, bits, sep;
 struct node *lchild, *rchild, *main_ptr;

 printf("entered restore branch \n");

depth = find_depth(root_ptr -> middle_ptr);
if (depth == 1)
 {
  bits = root_ptr -> bits_no / 4;
  sep = 3;
```

```
  if (root_ptr -> middle_ptr -> left_ptr != NULL)
    {
        main_ptr = root_ptr -> middle_ptr -> left_ptr;
        lchild = main_ptr;
        rchild = restore_child(lchild, bits, sep);
        root_ptr -> left_ptr -> left_ptr = lchild;
        root_ptr -> right_ptr -> left_ptr = rchild;
    }

  if (root_ptr -> middle_ptr -> right_ptr != NULL)
    {
        main_ptr = root_ptr -> middle_ptr -> right_ptr;
        lchild = main_ptr;
        rchild = restore_child(lchild, bits, sep);
        root_ptr -> left_ptr -> right_ptr = lchild;
        root_ptr -> right_ptr -> right_ptr = rchild;
    }
  root_ptr -> middle_ptr = NULL;

 } /* end of depth == 1 */

if (depth == 2)
 {
  bits = root_ptr -> bits_no / 8;
  sep = 9;

  if (root_ptr -> middle_ptr -> left_ptr != NULL)
```

```
  {
   if (root_ptr -> middle_ptr -> left_ptr -> left_ptr != NULL)
    {
     insert_node_if_reqd(root_ptr -> left_ptr, 0);
     insert_node_if_reqd(root_ptr -> right_ptr, 0);
        main_ptr = root_ptr -> middle_ptr -> left_ptr -> left_ptr;
        lchild = main_ptr;
        rchild = restore_child(lchild, bits, sep);
        root_ptr -> left_ptr -> left_ptr -> left_ptr = lchild;
        root_ptr -> right_ptr -> left_ptr -> left_ptr = rchild;
      }
   if (root_ptr -> middle_ptr -> left_ptr -> right_ptr != NULL)
    {
     insert_node_if_reqd(root_ptr -> left_ptr, 0);
     insert_node_if_reqd(root_ptr -> right_ptr, 0);
        main_ptr = root_ptr -> middle_ptr -> left_ptr -> right_ptr;
        lchild = main_ptr;
        rchild = restore_child(lchild, bits, sep);
        root_ptr -> left_ptr -> left_ptr -> right_ptr = lchild;
        root_ptr -> right_ptr -> left_ptr -> right_ptr = rchild;
      }
  } /* end of left branch*/

if (root_ptr -> middle_ptr -> right_ptr != NULL)
  {
   if (root_ptr -> middle_ptr -> right_ptr -> left_ptr != NULL)
    {
```

```
        insert_node_if_reqd(root_ptr -> left_ptr, 1);
        insert_node_if_reqd(root_ptr -> right_ptr, 1);
           main_ptr = root_ptr -> middle_ptr -> right_ptr -> left_ptr;
           lchild = main_ptr;
           rchild = restore_child(lchild, bits, sep);
           root_ptr -> left_ptr -> right_ptr -> left_ptr = lchild;
           root_ptr -> right_ptr -> right_ptr -> left_ptr = rchild;
          }
      if (root_ptr -> middle_ptr -> right_ptr -> right_ptr != NULL)
       {
        insert_node_if_reqd(root_ptr -> left_ptr, 1);
        insert_node_if_reqd(root_ptr -> right_ptr, 1);
           main_ptr = root_ptr -> middle_ptr -> right_ptr -> right_ptr;
           lchild = main_ptr;
           rchild = restore_child(lchild, bits, sep);
           root_ptr -> left_ptr -> right_ptr -> right_ptr = lchild;
           root_ptr -> right_ptr -> right_ptr -> right_ptr = rchild;
          }
     } /* end of right branch*/
    root_ptr -> middle_ptr = NULL;


  } /* end of depth == 2*/
} /* end of function*/
```

```
/* modify synthesized tree pointed to by "root_ptr" to make it suitable
for mux implementation*/

void modify_tree(struct node *root_ptr)
{
  int i, stat;



  if (root_ptr == NULL) return;
  if (root_ptr -> signal < -1) return;
  if ((root_ptr -> right_ptr == NULL)&&(root_ptr -> left_ptr == NULL))
    { modify_tree(root_ptr -> middle_ptr);
     return;
    }
  else
    if ((root_ptr -> signal == -1) && (root_ptr -> middle_ptr != NULL))
      {
       stat = determine_depth(root_ptr);
       if (stat == 0) switch_level(root_ptr);
       else
          { stat = determine_depth(root_ptr -> left_ptr) +
         determine_depth(root_ptr -> right_ptr) +
         determine_depth(root_ptr -> middle_ptr);

if (stat == 0)
          {
```

```
            switch_level(root_ptr ->left_ptr);
            switch_level(root_ptr ->right_ptr);
            switch_level(root_ptr -> middle_ptr);
            switch_level(root_ptr);
          }
         else if (stat > 0) restore_branch(root_ptr);
        }


    }

    modify_tree(root_ptr -> left_ptr);
    modify_tree(root_ptr -> right_ptr);

}



/* map synthesized tree pointed to by "root_ptr" into ACT cells
  returns the number of ACT cells required*/
int cellmap(struct node *root_ptr, int level)
{
  int i=0, newlevel;
if ((root_ptr == NULL) || (root_ptr -> signal < -1) ||
        (root_ptr -> bits_no < 4))
        return 0;
```

```
if ((root_ptr -> left_ptr == NULL) &&
        (root_ptr -> right_ptr == NULL) &&
        (root_ptr -> middle_ptr == NULL)) return 0;
  if ((root_ptr == NULL) || (root_ptr -> signal < -1) || (root_ptr -> bits_no < 4))
        return 0;
if ((root_ptr -> left_ptr == NULL) &&
        (root_ptr -> right_ptr == NULL) &&
        (root_ptr -> middle_ptr != NULL))
        return cellmap(root_ptr -> middle_ptr, level);


  if ((level % 2 == 0) || (root_ptr -> main_brother == 1))
        {
          root_ptr -> newcell = 1;
          newlevel = 1;
          printf("newcell %d \n", root_ptr -> node_id);
        }
  else
        {
          root_ptr -> newcell = 0;
          newlevel = level + 1;
        }
i = cellmap(root_ptr -> right_ptr, newlevel) +
cellmap(root_ptr -> left_ptr, newlevel) + root_ptr -> newcell;
return i;
}
```

```
main()
{
  struct node *main_root_ptr;
  int i, j, bit, count, numcells = 0, numcells2, max_level_length = 0, fixed_sig_no = 0;
  char line[81];
  temp_file = fopen("temp", "w");

  main_root_ptr = malloc(sizeof(struct node));
  if (main_root_ptr == NULL)
  {
    printf("malloc failed\n");
    return;
  }
  main_root_ptr -> node_id = 1;
  main_root_ptr -> var_no = 1;
  main_root_ptr -> signal = -1;

  fgets(line, 81, stdin);
  sscanf(line, "%d", &count);
  if (count <= 32)
  {
    (main_root_ptr -> bitmap1).bit_str = 0;
    (main_root_ptr -> bitmap2).bit_str = 0;
    for (i = count - 1; i > -1; i--)
```

```
    {
      fgets(line, 81, stdin);
      sscanf(line, "%d", &bit);
      if (bit == 1)
        (main_root_ptr -> bitmap1).bit_str |= (bit << i);
    }
  }
  else
  {
    (main_root_ptr -> bitmap1).bit_str_ptr = malloc(count / 32 *
                                        sizeof(int));
    (main_root_ptr -> bitmap2).bit_str_ptr = malloc(count / 32 *
                                        sizeof(int));
    for (i = 0; i < count / 32; i++)
    {
      *((main_root_ptr -> bitmap1).bit_str_ptr + i) = 0;
      *((main_root_ptr -> bitmap2).bit_str_ptr + i) = 0;
      for (j = 31; j > -1; j--)
      {
        fgets(line, 81, stdin);
        sscanf(line, "%d", &bit);
        if (bit == 1)
          *((main_root_ptr -> bitmap1).bit_str_ptr + i) |= (bit << j);
      }
    }
  }
  main_root_ptr -> bits_no = count;
```

```
  main_root_ptr -> subtree_id = 1;

  printf(" --- OUTPUT WITH THE INITIAL BITSTRING ---\n");
  for (i = 0; i < 1; i++)
  {
    main_root_ptr -> left_ptr = NULL;
    main_root_ptr -> right_ptr = NULL;
    main_root_ptr -> middle_ptr = NULL;
    main_root_ptr -> flip = 'n';
    create_tree(main_root_ptr, main_root_ptr, 2, &fixed_sig_no,
            &max_level_length);
    printf("Maximal level length is : %d\n", max_level_length);
   /*  printf("TREE :\n");
    traverse_tree(main_root_ptr);
    printf("\n");
    printf("MODIFIED TREE :\n");*/
    modify_tree(main_root_ptr);
   /* traverse_tree(main_root_ptr);*/
    printf("\n");
    numcells = cellmap(main_root_ptr, 0);
    printf("numcells1 = %d\n", numcells);
numcells2 = cellmap(main_root_ptr -> right_ptr, 0) +
                cellmap(main_root_ptr -> left_ptr, 0) + 1;

    if (numcells2 < numcells) numcells = numcells2;
    printf("numcells2 = %d\n", numcells2);
    fprintf(temp_file, "%d\n", numcells);
```

```
    }
  fclose(temp_file);
}
```

VITA AUCTORIS

Anthony Armah was born in Accra, Ghana. He graduated from high school in 1986. From there he went to the University of Ghana where he obtained a B. Sc in Computer Science and Physics in 1990. He then went to Japan where he obtained a M. Sc in the area of X-ray crystallographic studies at the Faculty of Pharmaceutical Sciences, Osaka University. He is currently a candidate for the Master's degree in Computer Science at the University of Windsor and will graduate in the Fall of 1998.